Constrained optimisation over massive databases

Toni Mancini[1], Pierre Flener[2], Amir Hossein Monshi[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; AmirHossein.Monshi.1351@student.uu.se

Abstract

Constrained optimisation is increasingly considered by industry as a major candidate to cope with hard problems of practical relevance. However, the approach of exploiting, in those scenarios, current Constraint or Mathematical Programming solvers has severe limitations, which clearly demand new methods: data is usually stored in potentially very-large databases, and building a problem model in central memory suitable for current solvers could be very challenging or impossible. In this paper, we extend the approach followed in [3], by presenting a declarative language for constrained optimisation based on SQL, and novel techniques for local-search algorithms explicitly designed to handle massive data-sets. We also discuss and experiment with a solver implementation that, working on top of any DBMS, exploits such algorithms in a way transparent to the user, allowing smooth integration of constrained optimisation into industrial environments.

1 Introduction

Constrained combinatorial optimisation is increasingly considered by industry as a major candidate to cope with hard problems of practical relevance, like scheduling, rostering, resource allocation, security, bio-informatics, etc. However, the approach of exploiting, in those scenarios, current Constraint (CP) or Mathematical Programming (MP) solvers has severe limitations, which clearly demand new methods:

- Problems of industrial relevance may easily exhibit sizes that are way *out of the reach* of current solvers: data usually resides on information systems, in forms of potentially *very large*, possibly *distributed relational databases* with complex integrity constraints, and cannot easily be transferred to central memory for being processed.
- Moving data into central memory also potentially affects *data integrity*. Solving very-large combinatorial problems is likely to need a huge amount of computing time, during which original data must be kept locked, thus freezing the normal activities of the information system.
- In several scenarios, the structure of both the data and the problem is very articulated and complex. Although it is always possible to map them into the constructs offered by current constraint modelling languages, this

Proceedings of the 16th International RCRA workshop (RCRA 2009):
Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion
Reggio Emilia, Italy, 11–12 December 2009

further step must be carefully designed and implemented, being a real engineering activity. This is usually perceived as a major obstacle by industry programmers, who usually do not have CP/MP and abstraction skills.

Several attempts to join combinatorial optimisation and databases have already been carried out in the literature, especially in the contexts of Deductive Database Systems (see, e.g., [9]) and Logic Programming (see, e.g., the system DLV^{DB} described in [11]).

As a different approach to cope with these issues and to provide effective means to spread combinatorial optimisation into industrial environments, in previous work [3] a non-deterministic extension of Relational Algebra (RA) has been proposed. This language, called NP-Alg, has the ability of expressing all problems in the complexity class NP, where (the decision versions of) many problems of practical relevance lie. NP-Alg has been the formal basis to derive an extension for the well-known database querying language SQL, with similar characteristics, but also able to handle optimisation problems. This second language, called ConSQL (i.e., SQL with constraints) proposes a new paradigm for modelling constrained optimisation problems, that of *modelling as querying*: a problem specification is given as a set of *views* devoted to encode solutions, and *constraints* that the contents of such views must satisfy. Of course, since we face with optimisation problems whose decision versions belong to the complexity class NP, the "views" we are talking about are not first-order definable, hence not expressible in standard SQL.

Advantages of this approach wrt the ones above are two-fold: *(i)* the modelling language is essentially SQL, plus a few new and very intuitive constructs: industrial programmers do not have to master CP/MP to formalise their problems; *(ii)* the paradigm allows solving approaches (like the one we consider in this paper) that interact transparently with the DBMS and its mechanisms for data transfers and access policies, and for maintaining data integrity, since the solving process, which is seen as a normal transaction by the DBMS, may easily recover from data updates.

In this paper, after recalling NP-Alg (Section 2) we make the following contributions:

1. We re-design our ConSQL language in order to obtain a language having very few deviations from standard SQL (Section 3).
2. We reconsider the local-search [6] approach suggested in [3], and argue that, to succeed in solving large problems, classical local-search has to be re-thought, with an explicit focus on the handling of massive data. In particular, we isolate two major issues in classical local-search and propose two methods to boost scalability: *dynamic neighbourhood reduction* (DNR), and *incremental evaluation of constraints* (IEC) (Section 4).
3. We provide a new implementation of the system, based on the ideas above and on a much wider portfolio of local-search algorithms, with full support to the user for declaratively choosing the search strategy to follow, and experimentally evaluate it against the original one in terms of performance and scalability (Section 5).

2 The language NP-Alg

NP-Alg [3] is a non-deterministic extension of RA, suitable for modelling combinatorial *decision* problems.

Syntax. Let $\mathbf{R}$ be a relational schema, i.e., a set of relations of arbitrary finite arities. An NP-Alg expression is a pair $\langle \mathbf{S}, \mathit{fail} \rangle$ where:

1. $\mathbf{S} = \{S_1^{(a_1)}, \ldots, S_n^{(a_n)}\}$ is a set of new relations of arbitrary finite arities $a_1, \ldots, a_n$. These relations are called "guessed". $\mathbf{R}$ and $\mathbf{S}$ are disjoint.
2. *fail* is an expression of plain RA on the new database vocabulary $\mathbf{R} \cup \mathbf{S}$.

Semantics. Given a finite database $\mathcal{D}$ over schema $\mathbf{R}$, let $DOM_{\mathcal{D}}$ be the unary relation representing the set of all constants occurring in $\mathcal{D}$. The semantics of an NP-Alg expression $\langle \mathbf{S}, \mathit{fail} \rangle$ is as follows:

1. For each possible extension of the relations in $\mathbf{S}$ with elements in $DOM_{\mathcal{D}}$, the expression *fail* is evaluated, using ordinary rules of RA.
2. If there exists one such extension for which *fail* evaluates to the empty relation $\emptyset$ (denoted as $\mathit{fail} \diamond \emptyset$), the *answer* to the decision problem is "yes". Otherwise, the answer is "no". When the answer is "yes", such extension of relations in $\mathbf{S}$ is a *solution* of the problem instance.

Intuitively, *fail* encodes *constraints* that extensions of $\mathbf{S}$ need to satisfy to be considered solutions. These are represented in terms of *violations*, hence the goal is to guess an extension of $\mathbf{S}$ that makes $\mathit{fail} = \emptyset$ (a dual approach, where constraints are defined in a 'positive' way can however be followed).

NP-Alg has strong relationships with existential second-order logic (ESO) [8] and first-order model-expansion (FO-MX) [7], with guessed relations playing the role of quantified predicates in ESO and expansion vocabulary symbols in FO-MX. It can be shown [3] that NP-Alg can express all (and only) the queries that specify decision problems belonging to NP.

In the following, we assume the standard version of RA [1] with the usual operators $\{\sigma, \pi, \times, -, \cup, \cap\}$, and positional notation ($\$1, \$2$, etc.) for attributes of the various relations. Also, given a tuple τ with n components, we denote its i-th component by $\tau[\$i], i \in [1\,..\,n]$, and the sub-tuple obtained from τ by considering only components with indices from i to j ($i \leq j$) by $\tau[\$i\,..\,\$j]$. Finally, to ease notation, especially when expressing conditions over tuples defined over the Cartesian product of some relations, we may denote tuple indices by the name of the original relations they come from, followed by the relative column number. Hence, e.g., $\sigma_{\$1=\$3}(R_1 \times R_2)$ is equivalent to $\sigma_{R_1.\$1=R_2.\$1}(R_1 \times R_2)$ if R_1 is binary.

Example 1 (Graph 3-colouring)**.** *Let $\mathcal{D}$ be a database with two relations N and E, encoding a graph: N is unary and encodes the set of nodes, while E is binary and encodes the set of edges of the graph, as pairs of nodes. Hence $DOM_{\mathcal{D}} = N$. The NP-complete Graph 3-colouring problem [5] can be expressed in* NP-Alg *by the following expression:*

1. $\mathbf{S} = \{R, G, B\}$ *all unary, encoding the set of nodes to be coloured in red, green, and blue, respectively;*
2. *fail*=*fail_cover*$\cup$*fail_disj*$\cup \underset{\$1}{\pi}$*fail_col, where: fail_cover*$= N - (R \cup G \cup B)$, *fail_disj*$= (R \cap G) \cup (R \cap B) \cup (G \cap B)$, *fail_col*$= \bigcup_{C \in \{R,G,B\}} \left(\sigma_{\$1 \neq \$2} (C \times C) \cap E \right)$.

For any guessed extension of R, G, B, fail_cover $= \emptyset$ iff every node is assigned at least one colour; similarly, fail_disj $= \emptyset$ iff every node is assigned at most one colour, and fail_col $= \emptyset$ iff no two adjacent nodes are assigned the same colour. Hence, fail$\diamond \emptyset$ iff there exist extensions for R, G, B encoding an assignment of colours to nodes such that no two adjacent nodes have the same colour. Such extensions are solutions to the problem instance.

3 ConSQL: modelling as querying

In this section we present a new and improved version of our ConSQL language [3]. ConSQL is a non-deterministic extension of SQL, with its optimisation-free subset having the same expressive power of NP-Alg. ConSQL is a super-set of SQL: users can hence safely exploit the rich set of language features of SQL during problem modelling. The current version of the language has only very few deviations from standard SQL, in the aim of greatly easing the modelling task of constrained optimisation problems for the average industry programmer.

A detailed description of the syntax and semantics of ConSQL is given in the appendix (available at `http://www.dis.uniroma1.it/~tmancini`). Here, we introduce the language 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 [4], in order to show the flexibility of the modelling language), namely finding a* schedule *for the lectures of a set of courses, in a set of classrooms, by* minimising *the total number of students that are enrolled in courses having overlapping lectures, by satisfying constraints about lecture allocation in time and rooms (*con1*,* con2*,* con6*), room equipment (*con3*), teachers' unavailability (*con4*) and conflicts (*con7*), and course lengths (*con5*).*

Data reside in a relational database with the tables listed in Table 1 (primary keys are underlined; foreign key constraints are omitted for brevity). A ConSQL *specification for this problem is as follows:*

```
create SPECIFICATION Timetabling (
  // A guessed view (see new construct CHOOSE), encoding a 'guessed' timetable, as an
  // assignment of courses to room/period pairs (w. some 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) CAN BE 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, relying on view ConflCourses
  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 and e2.course = cc.c2
```

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 nb. of lectures planned
Period(id, day, week, start, end)	Set of time-periods, plus information 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

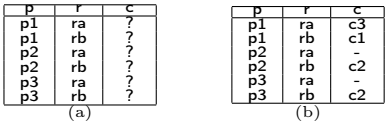
Table 1: Relational database schema for the University timetabling problem.

```
  // 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 nb of lectures for each course, in terms of helper view over TT
  create view CourseLengths as select c, count(*) as nb_lec from TT t where t.r is not null group by t.c
  check "con5" ( not exists (
    select * from CourseLengths cl, Course c where cl.c = 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 ))
);
```

As it 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) and a set of constraints (via the standard SQL check keyword). Some of the views are *guessed*, in that they have *special columns*, called *guessed column-sets*. Such columns are defined by the new construct CHOOSE, which is the main enabler of the non-deterministic mechanism added to SQL in order to increase its expressive power. The semantics is as follows:

- The solver is asked to *populate non-deterministically* the guessed column-sets of views, choosing tuples from the query argument of CHOOSE.
- This choice must be such that all *constraints* are satisfied and the *objective function*, if present, takes an optimal value.

In the example, our goal is to assign the lectures of some course to room/period pairs (guessed view TT), with some pairs possibly being empty (CAN BE NULL), in such a way that the number of students enrolled in courses with conflicting lectures is minimised and that all the constraints con1–con7 are



p	r	c
p1	ra	?
p1	rb	?
p2	ra	?
p2	rb	?
p3	ra	?
p3	rb	?

(a)

p	r	c
p1	ra	c3
p1	rb	c1
p2	ra	-
p2	rb	c2
p3	ra	-
p3	rb	c2

(b)

Figure 1: (a) Initialisation of guessed view TT on a sample database with 3 periods and 2 rooms. (b) A possible extension of TT if database has ≥ 3 courses ('-' denotes null.)

satisfied. The 'residual' (i.e., the pure SQL) part of view TT has schema (p, r), storing all period/room pairs. This is extended 'to the right' with a guessed column-set of width 1 having the schema (c) of the argument query. The overall view has schema (p, r, c). While the values in the first two columns are fixed (period/room pairs), those in the entries of the third column are non-deterministically picked from the sets of values occurring as ids for courses. Some of them can be assigned to null, allowing us to model situations, like ours, where we need *partial functions*. Figure 1(a) shows guessed view TT at the initialisation stage, when the database holds three periods with ids p1, p2, p3, and two rooms with ids ra and rb. Figure 1(b) shows one possible extension of the guessed view, when the database stores courses having ids c1, c2, c3 (plus, possibly, others). Note that, in case the modifier distinct were used, it would be impossible for two period/room pairs to be assigned to the same course. This is analogous to the *all-different* constraint used in CP.

Extensions of guessed views need to satisfy all constraints in order to be considered solutions to the problem. These are expressed by using the standard SQL keyword check applied to an arbitrary SQL Boolean expression (various forms allowed by standard SQL to specify conditions, hence constraints, are described in the appendix). Finally, an objective function can be defined by using one of the two new keywords MINIMIZE and MAXIMIZE applied to an aggregate arbitrary query returning a single scalar defined over database tables, ordinary views, guessed views.

4 Solving NP-ALG expressions over massive data

In principle, in order to solve a constrained optimisation problem, different competing techniques could be adopted, e.g., CP, Propositional Satisfiability (SAT), MP. In presence of massive data-sets, such *complete* approaches may show severe limitations, since they require the systematic exploration of the huge search space. In these cases, *incomplete* approaches like local-search [6], may help. These algorithms sacrifice completeness for efficiency, and may be advantageous especially on large instances, and in scenarios where computing a solution with a sufficient high quality is satisfactory. Local-search has been employed successfully to different formalisms, like SAT (cf. BG-WALKSAT [13]) and CP (cf., e.g., COMET [12]), the successful resolution of large scheduling and timetabling problems [10] or other large constrained optimisation problems [12].

In this section, after discussing some convenient syntactic extensions to NP-ALG, and recalling the main notions of classical local-search, we present novel techniques to make local-search applicable on massive data-sets.

NP-ALG revisited. The language described in Section 2 can be extended

without affecting its expressive power, with means to support bounded integers and arithmetic, typed guessed relations, guessed functions, and guessed permutations. As an example, although in the base version of the language entries of guessed relations take values ranging in $DOM_{\mathcal{D}}$ (hence extensions of an r-ary relation range over subsets of $DOM_{\mathcal{D}}^r$), in a (just syntactically) enriched version we could easily force them to be *typed* and also to model *functions* [3]. In this way we could easily model, e.g., the *Graph k-colouring* problem, where the set of the k available colours is given in the additional unary relation K, as follows:

Example 3 (Graph k-colouring)**.**

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

Guessed function Col is represented as a relation having two typed *columns,* \$1 *and* \$2*, with entries in* \$1 *ranging over N and those in* \$2 *ranging over K. Being also forced to be a total function, Col is such that any node in N occurs* exactly once *in the first column of Col. The expression for fail will evaluate to $\emptyset$ iff the guessed extension for Col is such that no two different nodes (*$\$1 \neq \$3$*) assigned to the same colour (*$\$2 = \$4$*) share an edge.*

It can be shown that the enriched language where guessed relations are *always* forced to be *typed total functions* between two *unary* database relations has the same expressive power as the base language. This is because: *(i)* tuples of non-unary relations can always be denoted by new constants which act as *keys*; and *(ii)* arbitrary guessed relations can always be encoded by characteristic functions. Also, the expression *fail* can always be rewritten in *disjunctive form*, as a *union* of sub-expressions. Since this variation is syntactically much closer to our target practical language ConSQL, in the remainder of this paper we will refer exclusively to it. Hence, from now on, we rely on the following definition:

Definition 1 (NP-Alg expression)**.** *An* NP-Alg *expression $\langle \mathbf{S}, fail \rangle$ 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 $S_i : D_i \to C_i$ forced to be a* total function *with domain D_i and co-domain C_i, with both these relations being* unary *and belonging to $\mathbf{R}$.*
2. *An expression $fail = \bigcup_{i=1}^{k} fail_i$ of plain RA on the vocabulary $\mathbf{R} \cup \mathbf{S}$.*

The expressions $fail_i$ encode *constraints* to satisfy, with their tuples for a given extension of the guessed functions representing *constraint violations.*

The gap between NP-Alg and ConSQL is now much smaller (see Example 3): guessed functions in NP-Alg neatly correspond to guessed column-sets of ConSQL, and expressions $fail_i$ correspond to `check not exists` constraints. Extensions like having multiple guessed column-sets in the same view, `null` values, and non-unary relations serving as domains and co-domains are not substantial, and could be simulated by means of additional joins, special values in the co-domain relations, and introducing unary key fields.

Classical local-search. The following notions play key-roles in all local-search algorithms. We recall them in terms of an NP-ALG expression $\langle \mathbf{S}, \mathit{fail} \rangle$ when evaluated over a finite database over schema $\mathbf{R}$.

Definition 2 (States, search space, costs, solutions)**.** *A* state*, or* candidate solution*, 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 sub-expressions* fail_i *when evaluated on* $\overline{\mathbf{S}}$*. A* solution *is a state of cost 0.*

Most of the local-search algorithms described in the literature [6] have, at their core, the following greedy behaviour:

current-state ← INIT-STATE()
while not TERMINATION-CONDITION() **do**
    move ← CHOOSE-IMPROVING-MOVE(*current-state*)
    if *move* ≠ NIL **then** *current-state* ← MAKE-MOVE(*move*) **else** BREAK

The idea is, after a –usually random– initialisation, to iteratively evaluate and perform small changes (*moves*) to the current state, in order to reduce the cost of the current candidate solution, until some termination condition (in terms of, e.g., the cost of *current-state* or the amount of computational resources used) is met. The universe of possible moves is chosen in advance by the programmer from the structure of the problem. Although this choice could in principle be arbitrary, in local-search moves are mostly defined to involve a very small change to the current state.

The simple structure of guessed functions in NP-ALG suggests a very natural definition for atomic moves, while any other possible definition could be given just by composition.

Definition 3 (Move)**.** *A triple* $\langle S,d,c \rangle$ *with* $S \in \mathbf{S}$ *(*$S : D \to C$*),* $d \in D$*,* $c \in C$*.*

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

The choice of moves to perform usually varies, depending on the particular algorithm considered. As an example, *gradient descent* chooses an improving move randomly, while *steepest descent* chooses the move that, if performed, *maximally* reduces the cost of the new 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 problems. A third very successful algorithm is *min-conflicts* that, once adapted to the structure of the search space in the NP-ALG framework, randomly selects one guessed function $S : D \to C$ and one of its domain values $d \in D$, and then chooses the best improving

move among all those regarding S and d. All these greedy algorithms return NIL if no moves lead to a cost reduction, and hence are easily trapped in the so-called *local minima*, i.e., states that have cost lower than all their *neighbours* (states that could be reached by executing a single move). To avoid being stuck in local minima, and to be able to continue search towards better states, usually these basic algorithms are enhanced with more sophisticated (not pure-greedy) techniques. As an example, *simulated annealing* executes also worsening moves, with a probability decreasing during time, while *tabu-search* dynamically filters the set of available moves, by forbidding those that would bring search back to recently visited states. Also, to increase robustness, such algorithms are enhanced with *random restarts*, joint together in *batches*, and/or wrapped by even more sophisticated *meta-heuristics* [6].

Classical local-search algorithms typically use problem constraints only to evaluate the quality of a move, i.e., its impact in the cost of the search state achieved if such a move is performed. In the context of constrained optimisation over massive data, this approach quickly becomes impractical.

Example 4 (University timetabling, continued)**.** *Assume that database relations have sizes* $|\mathsf{Room}| = r$, $|\mathsf{Period}| = p$, $|\mathsf{Course}| = c$. *By Definition 3, a steepest descent algorithm needs, at* any *step, to evaluate a constant number* prc *of candidate moves (each period/room pair can be reassigned to one of the other* $c-1$ *courses plus* null*) to find and choose the one that maximally reduces the cost of the new current state. By using a naive approach, for any such move, 7* SQL *queries, one for each constraint, should be evaluated on the new database obtained by changing the course assigned to one room/period pair.*

Even in not-so-large scenarios, e.g., $p = 960$ *(8 periods/day, 5 days/week for 6 months, observe that scheduling is not periodic, given, e.g., teachers' unavailabilities),* $r = 30$, $c = 40$, *the number of candidate moves to consider at each step is over* 10^6. *Now, assuming that each query takes only* 10^{-3} *seconds to run, and even ignoring time for modifying and rolling-back the database each time, a naive version of steepest descent would need about* $10^6 \cdot 7 \cdot 10^{-3} = 7 \cdot 10^3$ *seconds to execute one single move!*

Constraint-directed dynamic neighbourhood reduction (DNR). To overcome these difficulties, the concept of constraint-directed neighbourhoods has been recently proposed [2]. The idea is to focus on the problem constraints also to isolate *subsets* of the moves that possess some useful properties, such as those that are *guaranteed to be improving* wrt one or more constraints. These ideas can be exploited and extended in our context in a very elegant and natural way, thanks to the great simplicity of NP-ALG.

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 $\mathit{fail}_i = \sigma_{\phi_i}(S_{i_1} \times \cdots \times S_{i_s} \times R_{i_1} \cdots \times R_{i_r})$, with $S_{i_j} \in \mathbf{S}$ $(j \in [1\,..\,s])$ and $R_{i_p} \in \mathbf{R}$ $(p \in [1\,..\,r])$. In CONSQL this is equivalent to say that constraints can be expressed as not exists select-from-where queries, as it is often the case. Let us now focus on a single constraint i. Assume that, in a given state $\overline{\mathbf{S}}$, fail_i evaluates to a non-empty set of tuples V_i (that can be regarded as *violations* of constraint

i). In order to find a solution we need to *resolve* all violations of all constraints. Let us limit our attention on the extension $\overline{S} \in \overline{\mathbf{S}}$ of a distinguished guessed function S (binary relation encoding a function $S : D \to C$ with D, C unary relations in $\mathbf{R}$) occurring $m \geq 1$ times in the expression for $fail_i$. To emphasise the m occurrences of S, we can –wlog– rewrite $fail_i$ as follows (after also changing references to columns in ϕ_i accordingly):

$$fail_i = \sigma_{\phi_i}(S^{(1)} \times \cdots \times S^{(m)} \times \mathbf{T}), \tag{1}$$

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.

Since a move over S improving wrt constraint i must resolve at least one violation in $V_i \subseteq S^{(1)} \times \cdots \times S^{(m)} \times \mathbf{T}$, we give the following definition:

Definition 4 (Promising moves)**.** *The set of* promising moves *over guessed function S wrt constraint i is defined by the following RA query:*

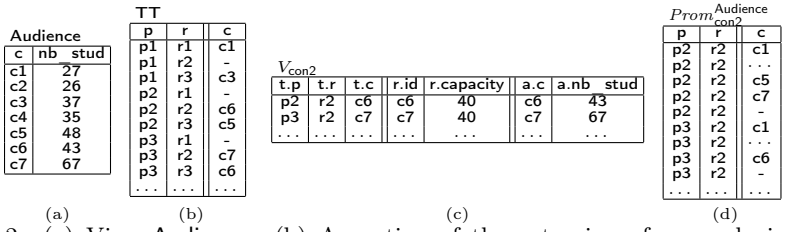
$$Prom_i^S = \underset{D,C}{\pi}\left(\underset{\chi \wedge \neg \phi_i'}{\sigma}(V_i \times D \times C)\right) \tag{2}$$

where $\chi = \bigvee_{j=1}^{m}(D.\$1 = V_i.\$(2j-1))$ and ϕ_i' is obtained from ϕ_i by replacing any atom a in which $S^{(x)}.\$2$ occurs ($x \in [1..m]$) with $((S^{(x)}.\$1 = D.\$1) \to a') \wedge ((S^{(x)}.\$1 \neq D.\$1) \to a)$ where a' is obtained from a by replacing any occurrence of $S^{(x)}.\$2$ with $C.\$1$.

Despite its apparent complexity, the meaning of formula (2) is quite intuitive. A generic tuple λ in V_i has the form $\langle d_1, c_1, \ldots d_m, c_m, \mathbf{t}\rangle$, with $d_1, \ldots, d_m \in D$ and $c_1, \ldots, c_m \in C$, and $\mathbf{t}$ a sub-tuple over schema $\mathbf{T}$. $Prom_i^S$ computes moves over S (hence pairs $\delta = \langle d, c\rangle \in D \times C$, cf. the projection operator) such that: *(i)* The domain value of δ, d, is equal to at least one among $d_1, \ldots d_m$ of at least one violation $\lambda \in V_i$ (condition χ); and *(ii)* For such a λ, tuple λ', built by replacing in λ any component c_j with the new value c iff the corresponding d_j is equal to d, does not satisfy condition ϕ_i. Since in RA we cannot modify tuples when evaluating a condition, in (2) we take the alternative approach of changing the condition itself into ϕ_i'. Requirement *(i)* guarantees that synthesised moves involve only domain values that appear in violations $\lambda \in V_i$, hence, if performed, resolve at least λ. Requirement *(ii)*, instead, ensures that, after performing δ, the newly introduced tuple in S, i.e., δ, does not lead to a new violation λ', at least with tuple $\mathbf{t}$. The following result holds (proofs omitted for space reasons):

Theorem 1. *In any state $\overline{\mathbf{S}}$, for any constraint i, and for any $S \in \mathbf{S}$ having extension $\overline{S}$ in $\overline{\mathbf{S}}$, all moves δ over S that, if executed from $\overline{\mathbf{S}}$, are improving wrt constraint i belong to $Prom_i^S$.*

In other words, since a move is improving iff it is improving wrt at least one constraint, by computing, in the current state, $Prom_i^S$ for all guessed functions S and all constraints i, we obtain a subset of the possible moves that contains *all* the improving moves, filtering out many (hopefully most) of the others. To better appreciate the advantages of this computation, let us reconsider the University timetabling problem.



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	c6	40	c6	43
p3	r2	c7	c7	40	c7	67
...	...	...	...	...	...	...

(c)

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

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	-
...	...	...

(d)

Figure 2: (a) View Audience. (b) A portion of the extension of guessed view TT in current state. (c) Portion of V_{con2} due to the given portion of TT. (d) Promising moves over TT computed starting from the displayed portion of V_{con2}.

Example 5 (University timetabling, continued)**.** *Assume that the problem instance refers to periods* p1–p40*, rooms* r1, r2, r3 *with capacities of 30, 40, 50 students, respectively, and courses* c1–c7*. The number of students enrolled in the various courses is given by view* Audience *in Figure 2(a). Suppose now that, in the current state, whose extension for* TT *is partially given in Figure 2(b), the expression for* con2 *evaluates to the set of violations* V_{con2} *of Figure 2(c). Tuples in* V_{con2} *give us precious information to drive the search. As an example, the first tuple shows that the current candidate timetabling has the problem that course* c6 *is given in period* p2 *in a room,* r2*, which is too small. Analogously for the other tuples. The query for* $Prom^{\mathsf{TT}}_{\mathsf{con2}}$ *(once generalised to handle domain and co-domain relations of arity greater than 1) returns the set of moves that resolve at least one violation in* V_{con2} *without producing a new one, at least with the same tuples of table* Room *and view* Audience*. Figure 2(d) shows part of the result of* $Prom^{\mathsf{TT}}_{\mathsf{con2}}$ *containing the moves synthesised from the first violation. Note that, by using the naive approach,* $prc = 40 \cdot 3 \cdot 7 = 840$ *moves needed to be considered. With the presented dynamic reduction instead, at most* 7 *moves* for each violation *of* con2 *need to be considered. Hence, the more progressed the greedy search, and the closer we are to a solution, the fewer the moves that need to be considered.*

DNR can be exploited also when we do not need knowledge of *all* improving moves. Consider, e.g., *min-conflicts*: at any iteration this algorithm randomly selects one guessed function $S : D \to C$ and one of its domain values $d \in D$, and then chooses the best improving move among all those regarding S and d. Given that for a move to be improving, it needs to resolve at least one violation of one 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, for each pair of components $\langle d_j, c_j \rangle \in \lambda$ caused by an occurrence of any guessed function $S_j : D_j \to C_j$ in the definition of the corresponding constraint, it considers as promising all moves of the kind $\langle S_j, d_j, c' \rangle, c' \in C_j, c' \neq c_j$, i.e., all moves that would make λ disappear in the target state. We omit a formal discussion of this variation for space reasons, but we will describe some preliminary experimental results in Section 5.

Incremental evaluation of constraints (IEC). Although DNR allows us to reduce the set of moves that need to be considered during a greedy search, it tells us nothing about the quantitative impact (in terms of cost change) of the surviving moves over i and the other constraints. To do this,

the naive algorithm would be to execute any such move and evaluate all problem constraints over the new database, then roll-back.

In the important and usual case of constraints of the form (1), the following results show that: *(i)* we can compute the exact cost variation upon all moves in a given set by running only 2 queries for each constraint; and *(ii)* after one such move has been chosen, we can update *incrementally* its impact in terms of violations added and resolved for all the constraints.

Definition 5 ($Resolved_i^S$ and $Added_i^S$)**.** *Given an* NP-ALG *expression* $\langle \mathbf{S}, fail \rangle$*, one of the guessed functions* $S \in \mathbf{S}$*, and one sub-expression* $fail_i$ *having the form* (1) *and evaluating to* V_i *in state* $\overline{\mathbf{S}}$*, let* M_S *be a relation over the schema of* S *encoding an arbitrary set of moves over* S*. We define the queries:*

$$Resolved_i^S = \underset{\chi'}{\sigma}\big(M_S \times V_i\big), \qquad Added_i^S = \underset{\xi}{\Psi}\Big(\underset{\chi'' \wedge \phi_i'}{\sigma}\big(M_S \times (S^{(1)} \times \cdots \times S^{(m)} \times \mathbf{T})\big)\Big),$$

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_i' *as in Definition 4. The operator* Ψ *in* $Added_i^S$ *is a non-standard* replacement *operator (that could easily be simulated in* SQL *by means of functions and* if *statements in the target-list) that changes each tuple* τ *of the input expression into* $\xi(\tau)$*. Here,* $\xi(\tau)$ *produces tuple* τ' *as follows:*

$$\forall i \in [1..m]\colon\ \tau'[S^{(i)}.\$1, S^{(i)}.\$2] = \begin{cases} \tau[M_S.\$1, M_S.\$2] & \text{if } \tau[S^{(i)}.\$1] = \tau[M_S.\$1] \\ \tau[S^{(i)}.\$1, S^{(i)}.\$2] & \text{otherwise} \end{cases}$$

while for any other component $\$j$ *of* τ*, we have that* $\tau'[\$j] = \tau[\$j]$*.*

These queries play a key-role in the incremental evaluation of constraints:

Theorem 2 (Incremental evaluation of constraints)**.** *Given* S*,* V_i *and* M_S *as in Definition 5, in the state* $\overline{\mathbf{S}} \oplus \delta$ *reached after performing any move* $\delta \in M_S$ *from state state* $\overline{\mathbf{S}}$*, the expression* $fail_i$ *for each constraint* i *evaluates to:*

$$\big(V_i - \underset{-M_S}{\pi}\big(\underset{M_S=\delta}{\sigma}(Resolved_i^S)\big)\big) \ \cup \ \underset{-M_S}{\pi}\big(\underset{M_S=\delta}{\sigma}\big(Added_i^S\big)\big),$$

with π_{-M_S} *denoting the projection operator that filters out columns due to* M_S*. Also,* $Resolved_i^S \cap Added_i^S = \emptyset$*.*

Theorem 2 tells us that, given an arbitrary set of moves over a guessed function S that occurs in the expression $fail_i$, we can compute *incrementally* the result of $fail_i$ in all the states reached by performing each of these moves, given the result in the current state. In particular, by taking $M_S = Prom_i^S$, we can compute the exact impact, both in terms of cost variations and in the extension of V_i, of all promising moves.

Query $Resolved_i^S$ returns a set of tuples, each representing a move $\delta = \langle d, c \rangle$ in M_S concatenated with a violation in V_i in which value d occurs in the first column of at least one $S^{(i)}$. Changing the co-domain value associated to d to c will make this violation disappear from V_i in the target state. As for $Added_i^S$, it computes the contributions, in terms of added tuples to the result of the query $fail_i$, of the execution of each move $\delta \in M_S$. Such

p	r	c	t.p	t.r	t.c	r.id	r.cap	a.c	a.nb stud
p2	r2	c1	p2	r2	c6	c6	40	c6	43
p2	r2	c5	p2	r2	c6	c6	40	c6	43
p2	r2	c7	p2	r2	c6	c6	40	c6	43
p2	r2	-	p2	r2	c6	c6	40	c6	43
p3	r2	c1	p3	r2	c7	c7	40	c7	67
p3	r2	c6	p3	r2	c7	c7	40	c7	67
p3	r2	-	p3	r2	c7	c7	40	c7	67

(a)

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

(b)

Figure 3: $Resolved^{\mathsf{TT}}_{\mathsf{con2}}$ (a) and $Added^{\mathsf{TT}}_{\mathsf{con2}}$ (b) for some moves in $Prom^{\mathsf{TT}}_{\mathsf{con2}}$ of Figure 2(d).

contributions are tuples of the Cartesian product in which value d occurs at least once (condition χ''), and 'modified' by properly taking into account the new value c that the moves under consideration assign. Again, since in RA we cannot modify tuples under evaluation, we take the dual approach of changing the condition from ϕ to ϕ'. However, since we also must return the modified tuples, we need to use the non-standard Ψ operator, which changes the tuples in the input relations in the same way as done by ϕ'.

Moreover, given that $Resolved_i^S \cap Added_i^S = \emptyset$, just by grouping and counting tuples in those results, *we can compute with one additional query the exact cost of all moves in set M_S over the constraints.* Also, if all V_is are materialised, we obtain *incremental maintenance* of the results of the corresponding $fail_i$s by deleting and inserting tuples returned by these queries.

Example 6 (University timetabling, continued)**.** *Figure 3 shows the results of computing queries $Resolved^{\mathsf{TT}}_{\mathsf{con2}}$ (part (a)) and $Added^{\mathsf{TT}}_{\mathsf{con2}}$ (part (b)) regarding the subset of moves in $Prom^{\mathsf{TT}}_{\mathsf{con2}}$* explicitly *shown in Figure 2(d). As an example, move* $(\mathsf{p2}, \mathsf{r2}, \mathsf{c1})$ *will reduce the cost of* con2 *by 1 (since it occurs once in $Resolved^{\mathsf{TT}}_{\mathsf{con2}}$, see Figure 3(a), and it does not occur in $Added^{\mathsf{TT}}_{\mathsf{con2}}$, see Figure 3(b)). Analogously, move* $(\mathsf{p2}, \mathsf{r2}, \mathsf{c5})$ *proves to be neutral wrt* con2.

Some brief comments follow on the complexity of the three queries. For each guessed function S and constraint i, and assuming that V_i is materialised and incrementally updated thanks to results of Theorem 2, we have that: *(i)* $Prom_i^S$ can be computed with two joins, the first of which, given the tight condition χ, does not lead to an explosion of the number of tuples (any tuple in V_i can match at most with m tuples of D, with m being usually very small); *(ii)* $Resolved_i^S$ involves a single join operation; *(iii)* $Added_i^S$ involves one more join operation than the expression of constraint i. Also in this case, the presence of the tight condition χ'' avoids in practice the combinatorial explosion of the number of tuples due to the added join.

5 Implementation, experiments, future work

We implemented a new ConSQL solver based on the ideas presented in Section 4, and built on top of a relational DBMS. The system, written in Java, takes as input a problem specification in ConSQL and uses standard SQL commands for choosing, evaluating, and performing moves, hence being independent of the DBMS used. The user can also annotate her specification with a declarative description of the search strategy to be used, given in terms of the local-search algorithms (and parameters) to run (gradient descent, steepest descent, tabu search, min-conflicts, simulated annealing) and their composition (via random restarts or batches). Details are given in the appendix.

In order to test the effectiveness of the techniques above on the scalability of the approach, we performed preliminary experiments, evaluating the performance gain of DNR+IEC with respect to the naive evaluation of moves and constraints. It is worth observing that our purpose was not to beat state-of-the-art solvers (and in fact *we do not even comment on satisfaction*), but rather to seek answers to the following questions: what is the impact of DNR+IEC on: *(i)* the reduction of the number of moves to evaluate; *(ii)* the overall performance of the greedy part of the search.

We experimented with two problems, Graph colouring and University timetabling, and two different greedy algorithms: steepest descent and the violation-directed version of min-conflicts mentioned in Section 4. The reasons behind our choices are as follows: steepest descent needs to evaluate, at every iteration, *all* possible moves, hence it is a good indicator of the impact of DNR+IEC in terms of effectiveness in neighbourhood reduction and the consequent performance gain. On the other hand, min-conflicts needs to evaluate a much smaller set of moves at any step: hence it can be much more efficient than the former, especially on large scale instances. However, we will see that also in this case DNR+IEC may play an important role.

Experiments have been performed on an Athlon64 X2 Dual Core 3800+ PC with 4GB RAM, using MySQL DBMS v. 5.0.75, with no particular optimisations. Instances of both problems have been solved running both algorithms with and without DNR+IEC. In order to perform fair comparisons, the two runs of the same algorithm on the same instance have been executed by using the same random seed. As for steepest descent, this implies that the two runs *started from the same state*, and executed the *same sequence of moves*, terminating at the *same local minimum*. This does not hold for min-conflicts, since when IEC is not activated, constraints are not materialised, and the order with which tuples are retrieved cannot be controlled without paying a very high price in terms of efficiency.

As for the choice of problems (specification are given in the appendix) and instances: specification for graph colouring is very simple and consists of one guessed column-set and one constraint (see Example 3). Given its compactness, it gives clean information about the impact of our techniques on a per-constraint basis. We used some quite large benchmark instances taken from `http://mat.gsia.cmu.edu/COLOR/instances.html`, looking for minimal colourings, with number of needed colours known in advance. As for University timetabling, in order to deal with real world instances, we changed our formulation to that of [4], where schedulings are periodic, and some data preprocessing is already done (e.g., constraint about equipment is already reduced to a relation listing rooms suitable for the various courses). Also, we had to ignore some constraints which would need non-conjunctive queries, not yet supported by DNR+IEC. The final formulation resulted in 8 ConSQL constraints (taking into account course lengths, room suitability, teachers' periodic unavailabilities, conflicts among courses with common students, and minimum course spread). As for the instances, we used some of those of Track 3 of the 2007 Intl. Timetabling Competition (`http://www.cs.qub.ac.uk/itc2007`), taken from `http://tabu.diegm.uniud.it/ctt`.

Some typical-case results are reported in Table 2. Enabling DNR+IEC considerably boosted performance: *impressive speed-ups were experienced on all instances*, especially when all moves need to be evaluated (steepest descent). Also in the cases where DNR+IEC runs did not terminate in the 1-hour timeout, the number of iterations executed is always much higher than with DNR+IEC disabled. However, DNR+IER brings advantages also when complete knowledge on the possible moves is not needed (min-conflicts), although speed-ups are unsurprisingly lower. Finally, we observed (see Figure 4) the *strong* expected *reductions of the size of the neighbourhood*, thanks to the action of DNR (steepest descent). In particular, DNR is able to filter-out often $> 30\%$ of the moves at the beginning of search, and *constantly more than 80%* (with very few exceptions) when close to a local minimum.

Given the promising, even if preliminary results, we are working on extending DNR and IEC to constraints defined by queries with aggregates and groupings, on evaluating them on other heuristics, and on investigating database optimisations, e.g., automated indices generation, in order to execute queries as efficiently as possible and tackle much larger scenarios, as those mentioned in Section 1.

Acknowledgements. This work has been partly carried out during visits of T. Mancini to Uppsala University, supported by the Swedish Foundation for Intl Cooperation in Research and Higher Education (STINT, grant KG2003-4723) and Blanceflor Foundation. P. Flener and J. Pearson are supported by grant 2007-6445 of the Swedish Research Council (VR).
Authors thank the anonymous reviewers for their comments and suggestions.

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] M. Cadoli and T. Mancini. Combining Relational Algebra, SQL, Constraint Modelling, and local search. *Theory and Practice of Logic Programming*, 7(1&2):37–65, 2007.
[4] 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'08*, 2008.
[5] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman & Co., 1979.
[6] H. H. Hoos and T. Stützle. *Stochastic Local Search, Foundations and Applications*. Elsevier/Morgan Kaufmann, 2004.
[7] 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.
[8] C. H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.
[9] R. Ramakrishnan and J. D. Ullman. A survey of deductive database systems. *J. of Logic Programming*, 23(2):125–149, 1995.
[10] A. Schaerf. A survey of automated timetabling. *Artif. Intell. Review*, 13(2):87-127, 1999.
[11] 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.
[12] P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. MIT Press, 2005.
[13] W. Zhang, A. Rangan, and M. Looks. Backbone guided local search for maximum satisfiability. In *Proc. of IJCAI 2003*, pages 1179–1186, 2003. Morgan Kaufmann.

Graph colouring				Steepest descent: sec (#iter)			Min-conflicts: sec (#iter)		
instance	nodes	edges	cols	DNR+IEC on	DNR+IEC off	speedup	DNR+IEC on	DNR+IEC off	speedup
anna	138	493	11	39 (21)	2682 (21)	69x	11 (14)	38 (33)	1.5x
games120	120	638	9	502 (40)	– (37)	8x	31 (48)	43 (48)	1.4x
homer	561	1629	13	247 (20)	– (2)	146x	36 (22)	53 (22)	1.5x
le450_5a	450	5714	5	– (140)	– (1)	140x	112 (22)	1667 (240)	1.4x
miles1000	128	3216	42	286 (38)	– (1)	478x	418 (25)	1266 (60)	1.3x
miles250	128	387	8	10 (28)	1734 (28)	173x	6 (10)	20 (36)	0.9x
miles500	128	1170	20	38 (28)	– (7)	379x	41 (14)	173 (46)	1.3x
miles750	128	2113	31	143 (36)	– (2)	453x	199 (25)	479 (46)	1.3x
mulsol.i.1	197	3925	49	– (4)	– (0)	?	637 (36)	1083 (47)	1.3x
mulsol.i.2	188	3885	31	2683 (60)	– (1)	81x	235 (22)	879 (64)	1.3x
mulsol.i.3	184	3916	31	1506 (54)	– (1)	129x	296 (28)	1228 (91)	1.3x
mulsol.i.4	185	3946	31	974 (56)	– (1)	207x	288 (27)	1238 (91)	1.3x
mulsol.i.5	186	3973	31	1827 (53)	– (1)	104x	146 (13)	991 (72)	1.2x
myciel6	95	755	7	14 (42)	1214 (42)	87x	10 (39)	19 (47)	1.6x
queen11_11	121	3960	11	152 (52)	– (9)	137x	36 (14)	200 (63)	1.2x
queen13_13	169	6656	13	– (36)	– (2)	18x	58 (8)	896 (104)	1.2x
zeroin.i.2	211	3541	30	287 (53)	– (1)	665x	274 (26)	917 (68)	1.3x

University timetabling					Min-conflicts: sec (#iter)		
instance	courses	lectures	rooms	periods	DNR+IEC on	DNR+IEC off	speedup
comp01	30	160	6	30	410 (5)	– (6)	7x
comp02	82	283	16	25	– (8)	– (2)	4x
comp03	72	251	16	25	– (12)	– (2)	6x
comp04	79	286	18	25	– (10)	– (1)	10x
comp05	54	152	9	36	511 (8)	– (11)	5x
comp06	108	361	18	25	– (2)	– (0)	?
comp07	131	434	20	25	– (1)	– (0)	?
comp08	86	324	18	25	– (8)	– (1)	?
comp09	76	279	18	25	3079 (15)	– (2)	9x
comp10	115	370	18	25	– (1)	– (0)	?
comp11	30	162	5	45	– (23)	– (4)	6x
comp12	88	218	11	36	– (16)	– (3)	5x
comp13	82	308	19	25	– (17)	– (0)	?
comp14	85	275	17	25	– (3)	– (1)	3x
comp15	72	251	16	25	– (10)	– (1)	10x
comp16	108	366	20	25	– (1)	– (0)	?
comp17	99	339	17	25	– (3)	– (1)	3x
comp18	47	138	9	36	142 (40)	– (47)	21x
comp19	74	277	16	25	1621 (8)	– (2)	9x
comp20	121	390	19	25	– (1)	– (0)	?
comp21	94	327	18	25	– (1)	– (0)	?

Table 2: ConSQL results in running steepest descent and min-conflicts on Graph colouring (top) and University timetabling (down), with and without DNR+IEC (1-hour timeout). Speedup = $(iter(on)/time(on))/(iter(off)/time(off))$. Since the size of university timetabling instances did never allow steepest descent to perform more than one iteration in 1 hour, such data has been omitted. With a longer timeout, we observed behaviours similar to those of Graph colouring.

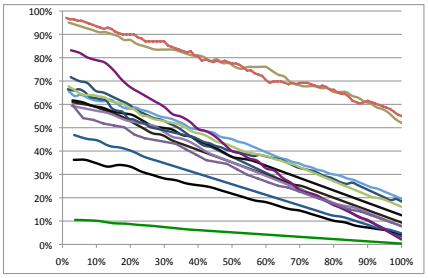


Figure 4: Ratio of number of promising moves wrt total number of moves, as a function of the state of run (0%=start, 100%=local min. reached) for all graph colouring instances in Table 2(top) for which the steepest descent run with DNR+IEC terminated in 1 hour.