Feb. 14, 2007: Submission to Workshop on Experimental Computer Science (San Diego, 13-14 June 2007)

Performance Testing of Combinatorial Solvers With Isomorph Class Instances

Franc Brglez
Dept. of Computer Science
NC State University
Raleigh NC, USA
brglez@ncsu.edu

Jason A. Osborne
Dept. of Statistics
NC State University
Raleigh NC, USA
osborne@stat.ncsu.edu

ABSTRACT

Combinatorial optimization problems that may be expressed as 'Boolean constraint satisfaction problems' (BCSPs) are being solved by different communities under different formulations and in different formats. If results of experimentation are reported, these can be seldom compared and replicated.

We propose a pragmatic approach to reconcile these issues: (1) use the familiar LP model that naturally expresses the constraints as well as the goals of the optimization task to formulate an optimization instance, (2) assemble and translate a number of hard-to-solve instances from different domains into the .lpx format parsed by at least two BCSP solvers: *lp_solve* in public domain, and *cplex*, (3) expose the intrinsic variability of BCSP solvers by constructing instance isomorphs as an equivalence class of randomized replicas of a reference instance; (4) use isomorph classes for the design of reproducible experiments with BCSP solvers that includes performance testing hypotheses; (5) release (on the web) all data sets, reported results, and software utilities used to prepare the data, invoke experiments, and post-process the results.

1. INTRODUCTION

Combinatorial optimization problems that may be expressed as 'Boolean constraint satisfaction problems' (BCSPs) [1] are being solved by different communities under different formulations and in different formats. If results of experimentation are reported, these can be seldom compared and replicated. An instance of a Boolean constraint satisfaction problem is given by m constraints applied to n Boolean variables. The well-known *conjunctive-normal-form* format (.cnf) captures such constraints very simply. However, different computational problems arise not only from the nature of constraints but also depend on the goals of the optimization task – a feature that is not supported by the .cnf format.

We propose a pragmatic approach to reconcile these issues:

- use the familiar LP model that naturally expresses the constraints as well as the goals of the optimization task when formulating an optimization instance; the .lpx format that expresses these constraints transparently is already parsed by at least two BCSP solvers: *lp_solve* [2] in public domain, and *cplex*, a state-of-the-art solver available under a commercial license [3].
- assemble and translate a number of hard-to-solve instances from different domains into the .lpx format and report runtime and best objective results obtained with the latest version of *cplex*;
- expose the intrinsic variability of BCSP solvers by constructing instance isomorphs as an equivalence class of randomized replicas of a reference instance;
- use isomorph classes for the design of reproducible experiments with BCSP solvers that includes performance testing hypotheses;
- release (on the web) all data sets, reported results, and software utilities used to prepare the data, invoke experiments, and post-process the results.

For years, publications on special purpose BCSP solvers have been comparing their performance to *cplex* whose performance was usually dominated by the new special-purpose solver being published. However, our recent work and comparisons with *cplex* reveals cases where *cplex* appears to dominate on a number of instances [4]. It is a given that the developer of a special purpose BCSP solver expects to design it in a way that will outperform a general purpose LP solver such as *cplex* which may only handle BCSPs on the side. One of the most important goals of this paper is to initiate a methodology of performance testing that will reliably measure and improve the performance of any and all BCSP solvers, thereby extending the work initiated in [5].

The paper is organized as follows. Section 2 introduces several classes of the Boolean constraint satisfaction problem (BCSP) under the and 0/1 integer program (IP) formulation, concluding with examples that provide a lead-in for the Section 3 on instance isomorph classes. Statistical techniques in Section 4, including hypothesis testing, are illustrated by running experiments with *cplex* on different classes of isomorphs. Compositions of instance blocks of increasing size, each with a 'hidden solution', are subject of Section 5. Data sets, all in .lpx format, and additional experimental designs are presented in Section 6. The paper concludes with an Appendix that begins with two small example of files in .lpx format, and proceeds by outlining features of the software utilities used to prepare the data sets (including a number of translators to/from .lpx), invoke experiments, and post-process the results.

2. INSTANCE FORMULATIONS

We start with basic notation and definitions and and conclude with examples that illustrate them.

Notation and Definitions. The combinatorial optimization problem is represented as a maximization problem in [6]:

$$\max\ \mathrm{w^T x} \quad \text{subject to} \quad \mathrm{Ax \geq b}, \quad \mathrm{x} \in \{0,1\}$$

where w is an n-vector in R_+^n or Z_+^n, b is an n-dimensional vector of 1's, and A is an $m \times n$ constraint matrix with entries from $\{0,1\}$. The minimization problem is represented similarly, with $Ax <= b$ (also known as set packing constraint) changed to $Ax >= b$ (also known as set cover constraint).

The 0/1 IP formulation is seldom used in textbooks on optimization of electronic system design [7, 8], these textbooks would refer to the formulation above as *unate*, to differentiate it from the more general *binate* formulation. In contrast to the unate formulation, the binate formulation includes positive and negative variables. Now, we show that both the maximization and the minimization instance can be always expressed with the '>=' relation, i.e.

$$\max\ \mathrm{w^T x} \quad \text{subject to} \quad \mathrm{Ax \geq b}, \quad \mathrm{x} \in \{0,1\}$$

and

$$\min\ \mathrm{w^T x} \quad \text{subject to} \quad \mathrm{Ax \geq b}, \quad \mathrm{x} \in \{0,1\}$$

where A is now an $m \times n$ constraint matrix with entries from $\{0, 1, -1\}$ and b is an n-dimensional vector whose entries are no longer 1's by default. The entries in b depend on the context of the constraint and also on the distribution of the $\pm$ signs within the constraint, as we explain next.

Denoting I_p and I_n as subsets of $\{1\ 2\ \ldots\ n\}$, we distinguish between three classes of constraints:

unate-positive, equivalent to the set cover constraint:

$$\sum_{i \in I_p} (+x_i) >= +1$$

i.e. at least one x_i must be set to 1.

unate-negative, equivalent to the set packing constraint:

$$\sum_{j \in I_n} (-x_j) >= -1$$

i.e. at most one x_j can be set to 1. Whenever $|I_n| > 2$, it defines a *clique constraint* [6] and can be decomposed into $|I_n|(|I_n| - 1)/2$ equivalent constraints. For example, the single constraint $-x_1 - x_2 - x_3 >= -1$ is equivalent to the following pair-wise constraints:
$-x_1 - x_2 >= -1$, $-x_1 - x_3 >= -1$, $-x_2 - x_3 >= -1$.

binate, a combination of set cover and packing constraints with a relaxed right-hand-side:

$$\sum_{i \in I_p} (+x_i) + \sum_{j \in I_n} (-x_j) >= +1 - |I_n|$$

If $I_p \in \emptyset$, the constraint $\sum_{j \in I_n} (-x_j) >= 1 - |I_n|$ is satisfied for all combinations of values of x_j, except for all $x_j = 1$.

If all constraints are unate-positive, the solution of the maximization instance is trivial, similarly for the minimization of the instance where all constraints are unate-negative. However, for the general case, both the maximization and the minimization can be equally hard.

REMARK: An instance of a Boolean constraint satisfaction problem (BCSP) is a maximization *or* a minimization problem with any combination of unate-positive, unate-negative, and binate constraints. Minimum (weighted) binate set cover, maximum (weighted) unate set packing, minimum (weighted) vertex cover, (weighted vertex) maximum clique, etc. are all BCSPs. Min Ones and Max Ones problems are special cases of unit-weighted BCSPs. Classes of Max CSP (Min CSP) problems as defined in [1] are also included in this formulation of BCSP. The next few example illustrate the structure of some such instances.

Instance examples. We show small examples and solutions of a weighted minimum set cover instance, a weighted vertex maximum clique instance that is derived directly from the structure of the set cover instance, and a weighted binate instance with a maximization objective. We also show solutions of related instances with the same structure: a weighted maximum set packing instance and a weighted binate instance with a minimization objective. Examples of additional instance transformations (and how they may relate) will be introduced in the full-length paper.

<u>A weighted minimum set cover instance.</u>
ObjectiveOpt 70
Solution 1010100
Min
$+21x_1 + 22x_2 + 23x_3 + 25x_4 + 26x_5 + 27x_6 + 29x_7$
st

$$\begin{array}{llllllllll}
c1: & & +x_2 & +x_3 & +x_4 & & & & >= & +1 \\
c2: & & +x_2 & & & +x_5 & +x_6 & & >= & +1 \\
c3: & & & & & +x_5 & +x_6 & +x_7 & >= & +1 \\
c4: & & & +x_3 & & & & +x_7 & >= & +1 \\
c5: & +x_1 & & & +x_4 & & & +x_7 & >= & +1 \\
c6: & +x_1 & & +x_3 & & & +x_6 & & >= & +1
\end{array}$$

A weighted maximum set packing instance.
This instance is generated from the set packing instance by (1) flipping the '+' variable signs in each row to '-', (2) replacing the right-hand-side with values of -1, and (3) changing the objective from 'min' to 'max'.
ObjectiveOpt 52
Solution 0001010

A weighted vertex maximum clique instance.
This instance is generated from the set packing instance by (1) expanding all clique constraints into pair constraints (one pair on each row), (2) flipping the '+' variable signs in each row to '-', (3) replacing the right-hand-side with values of -1, and (4) changing the objective from 'min' to 'max'.
ObjectiveOpt 100
Solution 1010011
Max

$$+21x_1 + 22x_2 + 23x_3 + 25x_4 + 26x_5 + 27x_6 + 29x_7$$

st

$$\begin{array}{lccccccccc}
c1: & & & -x_3 & & -x_5 & & & >= & -1 \\
c2: & & & & -x_4 & -x_5 & & & >= & -1 \\
c3: & & -x_2 & & & & & -x_7 & >= & -1 \\
c4: & & & & -x_4 & & -x_6 & & >= & -1 \\
c5: & -x_1 & & & & -x_5 & & & >= & -1 \\
c6: & -x_1 & -x_2 & & & & & & >= & -1
\end{array}$$

A weighted binate instance (obj=max).
ObjectiveOpt 100
Solution 0110101
Max

$$+21x_1 + 22x_2 + 23x_3 + 25x_4 + 26x_5 + 27x_6 + 29x_7$$

st

$$\begin{array}{lccccccccc}
c1: & & +x_2 & +x_3 & +x_4 & & & & >= & +1 \\
c2: & & -x_2 & & & -x_5 & -x_6 & & >= & -2 \\
c3: & & & & & +x_5 & +x_6 & -x_7 & >= & 0 \\
c4: & & & -x_3 & & & & +x_7 & >= & 0 \\
c5: & -x_1 & & & -x_4 & & & -x_7 & >= & -1 \\
c6: & -x_1 & & -x_3 & & & -x_6 & & >= & -1
\end{array}$$

A weighted binate instance (obj=min).
ObjectiveOpt 22
Solution 0100000
This instance is generated from the binate instance above by simply changing the objective from 'max' to 'min'.

3. CLASSES OF INSTANCE ISOMORPHS

Isomorphs of sat instances have been shown to induce significant variability in SAT solvers [5]. In this paper, we demonstrate that instance isomorphs of BCSP's (Boolean constraint satisfaction problems) as defined in the preceding section are also fundamental to exploring performance variability of combinatorial solvers that take them as input.

Given a (sparse) matrix formulation of the reference instance, an isomorph is generated by applying to the reference any subset of four primitive operations:

C: random permutation of variables – effectively a permutation of columns in the matrix;

L: random permutation of the variable order in any row of the matrix;

R: random permutation of rows in the matrix, followed by permutation of the weight vector (not needed if all weights have the value of 1);

X: random sign flipping (from positive to negative and vice versa) of any variable – while maintaining consistency of the right-hand-side value so that the instance remains a BCSP and the value of its objective function invariant.

The operation of flipping the variable sign (X) has intrinsic merits with SAT solvers and can only be applied to instances of BCSP in special situations. In this paper, we shall consider isomorphs in two equivalence classes only: LR and CLR. Two isomorphs from each of the two classes are shown below, based on LR operations and CLR operations applied to the same reference instance: the weighted binate instance in the previous section.

A weighted binate instance (obj=max) – isomorph_LR.
ObjectiveOpt 100
Solution 0110101
@VariablePermutationPairs (isomorph,reference – terminated with 0,0)
1,1 2,2 3,3 4,4 5,5 6,6 7,7 0,0
Max

$$+21x_1 + 22x_2 + 23x_3 + 25x_4 + 26x_5 + 27x_6 + 29x_7$$

st

$$\begin{array}{cccccc}
-x_3 & -x_1 & -x_6 & >= & -1 \\
-x_1 & -x_4 & -x_7 & >= & -1 \\
-x_5 & -x_2 & -x_6 & >= & -2 \\
+x_3 & +x_2 & +x_4 & >= & +1 \\
+x_7 & -x_3 & & >= & 0 \\
-x_7 & +x_6 & +x_5 & >= & 0
\end{array}$$

It is clear by inspection that no permutation of variables took place in the isomorph_LR, while rows have been permuted (row 1 in the reference instance is now row 4 in the isomorph). Furthermore, the order of variable positions in the row 4 in the isomorph is different from the order of variable positions in the row 1 in the reference instance.

On the other hand, column or variable permutation also took place in the isomorph_CLR below: if we know the permutation, the effort to verify that new new instance is in fact the isomorph of the reference is relatively simple.

A weighted binate instance (obj=max) – isomorph_CLR.
ObjectiveOpt 100
Solution 1100011
@VariablePermutationPairs (isomorph,reference – terminated with 0,0)
1,3 2,1 3,2 4,5 5,6 6,4 7,7 0,0
Max

$$+22x_1 + 23x_2 + 21x_3 + 27x_4 + 25x_5 + 26x_6 + 29x_7$$

st

$$\begin{array}{cccccc}
-x_3 & -x_7 & -x_5 & >= & -1 \\
+x_1 & +x_2 & +x_5 & >= & +1 \\
-x_2 & +x_7 & & >= & 0 \\
-x_3 & -x_2 & -x_4 & >= & -1 \\
-x_6 & -x_1 & -x_4 & >= & -2 \\
+x_4 & +x_6 & -x_7 & >= & 0
\end{array}$$

Since one may be tempted to dismiss LR-isomorphs as trivial, we bring forward a 350-variables example described in more detail later. The name of the isomorph class is f51mb-_350_B_40v_20_20_LR, and its reference instance is in cnf-format, i00.cnf. Since *cplex* takes files in .lpx format, we must translate it. The act of translation alone can induce instances in LR-class, depending on the implementation of the translator program. Let the first translator produce an instance in the 'reference order' given by the instance in the .cnf format and let two more translators rely on some hashing schemes that result in instances having row orders that are both different from the row order of the reference instance. Also, the order in which the variable appear in each row may be different. Such instances can be found in the class of 1+32 instances in the web-archive under the directory f51mb_350_B_40v_20_20_LR, say i00.lpx, i06.lpx, and i17.lpx. Upon invoking *cplex* 9.0 on each of these instance, we get a solution and a proof of optimality, however runtimes differ dramatically, despite running on the same dedicated CPU:

translator	instance	Obj_opt	RunTime (secs)
T1	i00.lpx	24	114.91
T2	i06.lpx	24	82.55
T3	i17.lpx	24	1801.86

These instances under f51mb_350_B_40v_20_20_LR do not represent the extreme cases: instance i12 is solved for the same optimum in 60.37 seconds, while instance i30 times out at 2115.28 seconds without proving that the best objective reported at 24 is indeed the optimum.

As shown in sections that follow, such solver sensitivity to the order of data in the instance file is not unusual – which explains why researchers may report vastly different performance results with the same instance, on the same platform, and with the same version of the solver!

Two questions arise: (1) do instances from a CLR-class induce solver variability that is equivalent to the variability induced by instance in the LR-class, and (2) is a CLR-isomorph class needed and why. The answer to the second question is affirmative – and is based on a few years of 'lessons-learned' experience [9, 10].

We do need to perform most if not all experiments with instances from the CLR-class because we cannot anticipate when we may encounter a 'smart solver' that will attempt to re-order input data in some predetermined fashion, so that most if not all instances from the LR-class may be re-ordered with relative ease into an almost equivalent if not equivalent order[1]. While this is apparently not the case (yet) with the *cplex* solver, we have had the experience with 'smart' BDD variable-ordering solvers where the only way to expose their sensitivity to order requires that we also permute the variables in each input file instance [10].

The first question can be rephrased as a formal hypothesis and resolved with standard statistical techniques, discussed in the following section. We will also show that the same technique can also be applied to resolve a related question: given experimental results from two solvers on randomly selected instances from a CRL-class and on the same platform, is the runtime performance of two solvers equivalent?

[1]Such strategy has also been demonstrated to backfire since it prevents the solver from 'seeing' many input orders that could improve its average performance.

4. ON STATISTICAL TECHNIQUES

The example with three isomorphs in preceding section motivates a formalized statistical approach to testing the performance of BCSP solvers. We thus expand the experiment from three instances in a LR-class in the previous section to a number of isomorph classes, with 32 randomly selected isomorphs in each class.

Initial Experiments. To initiate the experiments, we introduce seven isomorph classes that are derived from five reference instances as follows:

in201_cliq_CLR, where the reference instance in201_cliq.lpx represents a weighted-vertex maximum clique problem.

in201_cliq1_CLR, where the reference instance in201_cliq1.lpx represents a maximum clique problem related to the one above, except that all vertex weights have the value of 1.

alu4_CLR, where the reference instance alu4.lpx represents a minimum binate cover problem.

in401_sp_LR, where the reference instance in401_sp.lpx represents a weighted maximum set packing problem.

in401_sp_CLR, where the reference instance in401_sp.lpx is already defined above.

f51mb_0350_B_0040_20_20_LR, where the reference instance f51mb_0350_B_0040_20_20.lpx represents a specific block composition of two minimum binate cover problems.

f51mb_0350_B_0040_20_20_CLR, where the reference instance f51mb_0350_B_0040_20_20.lpx is already defined above.

For more information about each reference instance and the computing platform, see Table 1 in the section that follows.

For all instances in classes listed above, we run *cplex* as a branch&bound solver that reports the same the optimum value for each instance in its class – what is being observed is the *RunTime* to find this optimum. The results of these experiments are summarized in Figure 1. Tables in this figure report *RunTime* statistics for each class; note also that we report the runtime for each reference instance in a separate column *RefV*. We determine the reported distribution by running a combination of tests on the observed data: ranging from Cramer-Von Mises, Kolmogorov-Smirnov to χ^2 goodness-of-fit-tests [11, 12]. We also plot *empirical cumulative distribution functions* (ECDFs) for classes of most interest (LR vs CLR), and the barcharts that illustrate the runtime values for each isomorph in the respective LR and CLR classes. An itemized summary of our observations follows.

in201_cliq_CLR: the average runtime to solve instances in this class is only 3.42 seconds and the distribution is uniform.

***RunTime* statistics for two clique CLR classes and a binate cover CLR class.**

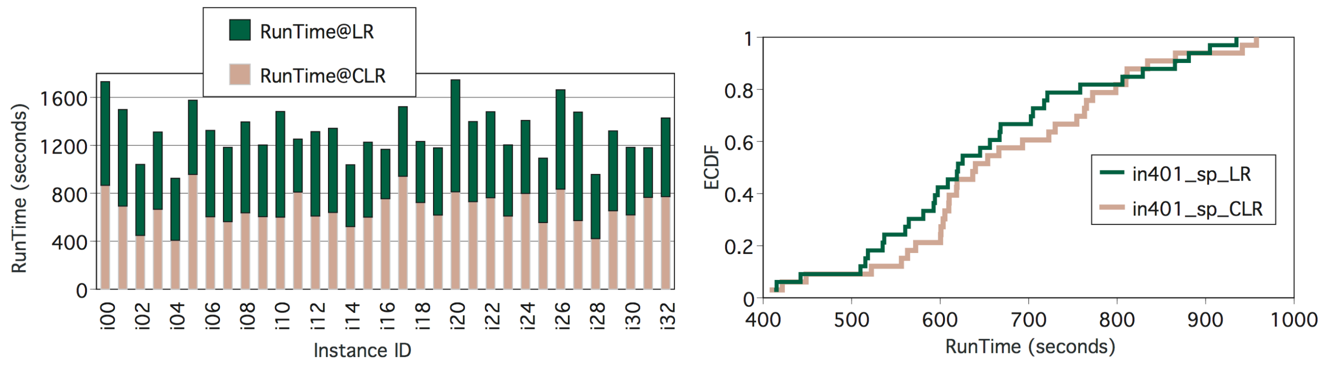
(*RefV* denotes the reference instance, excluded from the computation of min, max, median, mean, and standard deviation.)

Here, branch&bound solves each instance before time-out to an optimum value, then reports runtime.

Class	RefV	MinV	MaxV	MedV	MeanV	StdV	N	Distribution
in201_cliq_CLR@BB	3.56	3.1	3.76	3.45	3.42	0.17	32	uniform
in201_cliq1_CLR@BB	235	150	215	179	181	16.7	32	uniform
alu4_CLR@BB	38.5	23.5	1260	113	207	283	32	near-exponential

***RunTime* statistics for isomorph classes in401_sp_LR and in401_sp_CLR.**

Class	RefV	MinV	MaxV	MedV	MeanV	StdV	N	Distribution
in401_sp_LR@BB	865	412	935	620	639	133	32	uniform
in401_sp_CLR@BB	865	407	957	638	666	133	32	uniform



***RunTime* statistics for isomorph classes f51mb_0350_B_0040_20_20_LR and f51mb_0350_B_0040_20_20_CLR.**

Class	RefV	MinV	MaxV	MedV	MeanV	StdV	N	Distribution
f51mb_350_B_40_20_20_LR@BB	115	60.4	2115	110	256	458	32	heavy-tail
f51mb_350_B_40_20_20_CLR@BB	115	71.3	2118	127	232	393	32	heavy-tail

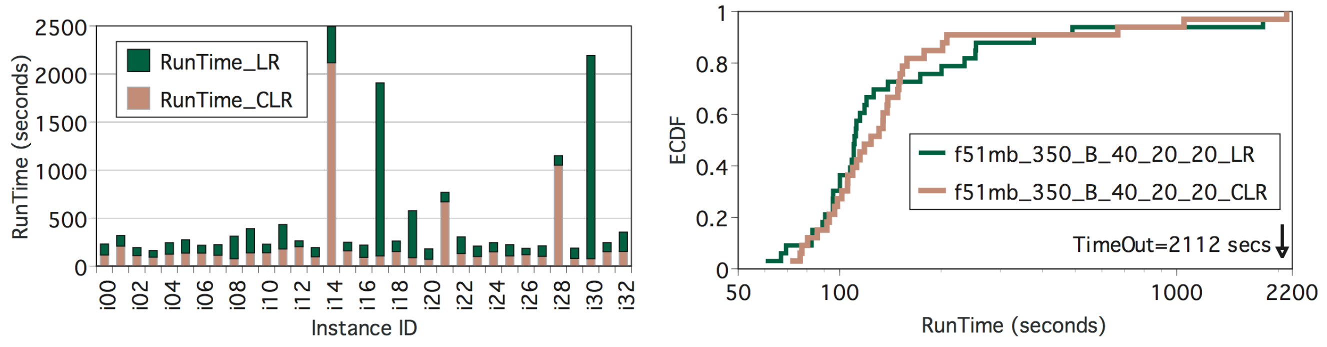


Figure 1: *Branch&bound* experiments with LR and CLR classes of isomorphs.

in201_cliq1_CLR: the average runtime to solve instances in this class is 181 seconds and the distribution is uniform. Given that the only difference between this and the previous class is that instances in the previous instance have non-unity weights, the presence of unity weights in this class is a factor that increases the 'difficulty' of this class significantly – when compared to the previous class.

alu4_CLR: the average runtime to solve instances in this class is 207 seconds, and the runtime ranges from 23.5 seconds to 1260 seconds. Instances in this class induce a near-exponential distribution for *cplex*.

in401_sp_LR: the average runtime to solve instances in this class is 639 seconds, and the distribution is uniform.

in401_sp_CLR: the average runtime to solve instances in this class is 666 seconds, and the distribution is uniform. Since this class is derived from the same reference instance as the previous class, the question arises if the two classes are equivalent, given the apparent "closeness" of the two ECDFs. We shall resolve this question with a hypothesis test shortly.

f51mb_0350_B_0040_20_20_LR, the average runtime to solve instances in this class is 256 seconds, and the distribution is heavy-tail.

f51mb_0350_B_0040_20_20_CLR: the average runtime to solve instances in this class is 232 seconds, and the distribution is heavy-tail. Since this class is derived from the same reference instance as the previous class, the question arises if the two classes are equivalent, given the apparent "closeness" of the two ECDFs. One more hypothesis test will be considered.

We state and resolve the following hypothesis:

$H0 : \mu_{LR} = \mu_{CLR}$
i.e. instances drawn at random from the LR-class are equivalent to instances drawn at random from the CLR-class. We test this hypothesis by finding the independent samples t-statistics, (the degrees-of-freedom in both cases: 32 + 32 - 2 = 62)
(in401_sp_LR, in401_sp_CLR) induces $t = 0.812 < 2.00$
(f51mb..._LR, f51mb..._CLR) induces $t = 0.224 < 2.00$
thus we we fail to reject the hypothesis at the 5% significance level.

Similarly, we can state a hypothesis about the average runtime performance of two solvers, A and B, on instances from an isomorph class CLR.

$H0 : \mu_A = \mu_B$ on instances drawn randomly from $\mathcal{I}_{CLR}$
Again, we test this hypothesis by finding the independent samples t-statistics, (the degrees-of-freedom: 32 + 32 - 2 = 62)
($A[\mathrm{I}_{CLR}]$, $B[\mathrm{I}_{CLR}]$) and test for $t < t_{crit}$
If the condition is met, we fail to reject the hypothesis at the 5% significance level.

Additional Experiments. We continue the experiments by introducing two classes of isomorphs as well as a collection of random instances that are claimed to bear some relationship to these isomorphs:

in401_sp_CLR, where the reference instance in401_sp.lpx represents a weighted maximum set packing problem with 1000 vertices and 1000 constraints.

in413_sp_CLR, where the reference instance in413_sp.lpx represents a weighted maximum set packing problem with 1000 vertices and 1000 constraints.

in401_sp_RND, where each instance in the set represent a randomly generated weighted maximum set packing problem with 1000 vertices and 1000 constraints.

See Table 1 for more information about the reference instances in401_sp.lpx and in413_sp.lpx.

Again, we run *cplex* as a branch&bound solver on all instances above. Now, the only random variable associated with the class in401_sp_CLR is *RunTime* since *ObjectiveBest* remains constant. However, since in413_sp_CLR is solved only 8 times and 25 instances time out at 1056 seconds, the random variables observed now are both *RunTime* and *ObjectiveBest.* It is obvious that instances in the class in413_sp_CLR are different from instances in the class in401_sp_CLR. Moreover, the differences from instance to instance are even more pronounced when we consider 32 instances from the 'class' in401_sp_RND. Again, both *RunTime* and *ObjectiveBest* are random variables, but now over significantly wider range than observed for the instances from either of the CLR classes above. An itemized summary of our observations follows.

in401_sp_CLR: the average runtime to solve instances in this class is 666 seconds and the distribution is uniform.

in413_sp_CLR: the average runtime to solve instances in this class is 1021 seconds and the distribution is *incomplete* due to too many timeouts.

in401_sp_RND: the average runtime to solve instances in this class is 894 seconds and the distribution is *incomplete* due to too many timeouts.

The only assertion we can make with certainty about the three classes discussed above is that the instances in the class in401_sp_RND are all very different from each other and that in401_sp_CLR and in413_sp_CLR represent two different classes of isomorphs; instances in in413_sp_CLR are much harder to solve.

5. BLOCK INSTANCE GENERATOR

An block instance generator has been designed to compose a structured block instance from a pair of instances. Optimal objective values and the solutions are presumed to be known for each instance. If such pair is composed into a block diagonal form with no addition of row constraints that would introduce a variable overlap between the two instances, the block instance has a known hidden solution as a concatenation of two solutions from each instance, with the optimum value of the new instance simply the sum of the the objective values for each instance. By following few simple rules, *we can maintain this additive property* even when we introduce overlap rows to the block instance. Recursively, we can create, in linear time, very large instances with specified overlap and with guaranteed hidden solutions that are optimal.

Details will be presented in the full-length paper. For the time being, we illustrate some aspect of the method with a partial response from the generator itself.

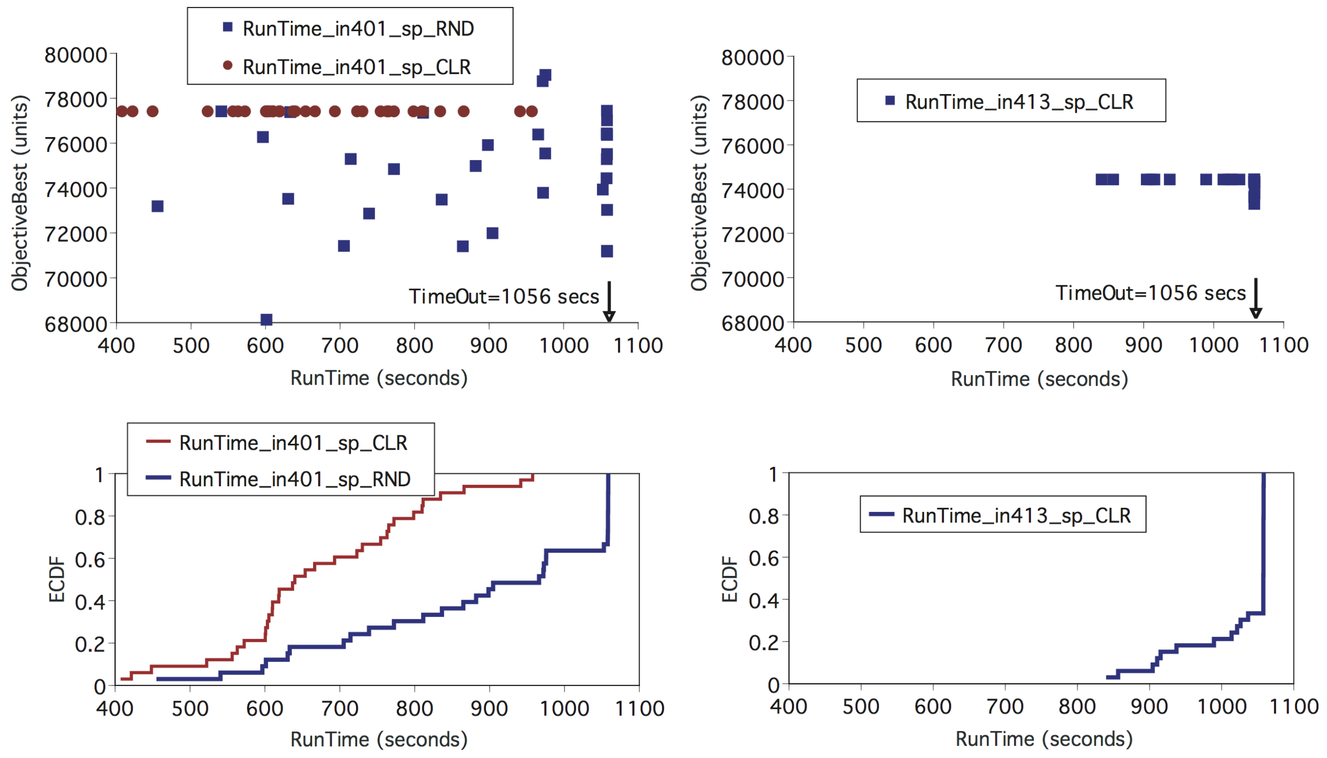
```
$ blocksDRS4lpx -info
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
              DESCRIPTION OF PROGRAM 'blocksDRS4lpx'
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

blocksDRS4lpx takes two *.lpx files with

  (n1-variables, m1-constraints), (n2-variables, m2-constraints)

concurrently with two *.BOUNDS files with same basenames. The
*.BOUNDS files contain ObjectiveOpt values and binary solution
strings for each instance.

The output is a *.lpx instance file with (n1+n2)-variables in a
block-diagonal form of at least (m1 + m2) rows. An overlap block
of additional rows with (n1+n2)-variables may be specified from
the command line. The method by which the overlap block is
generated is explained here by way of an example:

   n1 = 6 and n2 = 7, with solution strings 110100 and 1100001
```

***RunTime* statistics for the isomorph classes in401_sp_CLR, in413_sp_CLR and the random 'class' in401_sp_RND.**

(*RefV* denotes the reference instance, excluded from the computation of min, max, median, mean, and standard deviation.)

Here, branch&bound solves each instance class in401_sp_CLR to an optimum value of 77418.
However, a number of instances in in413_sp_CLR and in401_sp_RND time out at 1056 seconds.

Class	RefV	MinV	MaxV	MedV	MeanV	StdV	N	Distribution
in401_sp_CLR@BB	865	407	957	638	666	133	32	uniform
in413_sp_CLR@BB	1056	840	1056	1056	1021	65.7	32	incomplete
in401_sp_RND@BB	865	455	1056	969	894	177	32	incomplete



CAUTION: Not all statistics as reported in this figure are 'valid'. See also the body of the text.

(1) All the instances in the class in401_sp_CLR are well-defined as isomorphs and are solved by branch&bound solver under the time out value of 1056 seconds. Since the class is well-defined all optima have the same value (77418), the runtime distribution is thereby well-defined, and the statistics as reported for the class in401_sp_CLR are valid.

(2) Only 22 instances in the 'random class' in401_sp_RND time are solved by branch&bound under 1056 seconds; there are 22 distinct optima, ranging from 68135 to 79040 – i.e. these instances are *not in the same nor are they isomorphs.*
For the 11 instances that time out at 1056 seconds, values of *ObjectiveBest* range from 71170 to 77444. There are no indicators of how similar or different these instances really are. We label the distribution *incomplete* due to too many timeouts.

(3) When we take an instance in413_sp from the random class (in401_sp_RND) and create an isomorph class in413_sp_CLR, only eight instances in the isomorph class solve for an optimum value of 74435, a total of 25 instances time out at 1056 seconds (including the reference instance); values of *ObjectiveBest* for timed-out instances range from 73329 to 74435. This instance is different from the instance in401_sp – as are most if not all instances in the class in401_sp_RND. We label the distribution *incomplete* due to too many timeouts.

Figure 2: *Branch&bound* experiments with instances from an isomorph class and and a random 'class'.

```
that induce the following four lists of variables:

    Ones1 = (1 2 4)  Zeros1 = (3 5 6)
```

```
    Ones2 = (7 8 13) Zeros2 = (9 10 11 12)

With the option -rows=5,
  we get five pairs of unique unate constraints, chosen
  randomly from Ones1 and Ones2:

              (1 7) (4 8) (1 13) (2 7) (4 7)
....
....
```

6. DATA SETS AND MORE EXPERIMENTS

A substantial number of BCSP instances has been collected, translated into the .lpx format, and run in *cplex*. A subset of these instances and runs is summarized as *reference instances* in Table 1. A larger set and similar results are being prepared for a technical report and a web-posting under http://www.cbl.ncsu.edu/xBed/.

Table 1 summarizes instance categories and current status vis-a-vis *cplex*. As shown, most instance have not been solved optimally and represent an on-going challenge for *cplex* and other BCSP solvers. Here are some additional details.

min set cover (unate)
Instances ex5.pi and test4.pi represent column-row reduced versions of the most challenging unate instances from the LogicSyn91 set [13]. Instances in*_sc have been transformed into set cover instances from the set packing instances described below.

min set cover (binate)
Instances rot.b, alu4, e64.b represent column-row reduced versions of the most challenging binate instances from the LogicSyn91 set [13].

max set packing (unate)
Instances in*_sp are translated versions of set packing instances kindly submitted by Y. Guo, as a follow-up on a publication request [14], now updated in [15]. This a set of 500 random instances in five size categories, from 500 variables to 1500 variables. We adopted the first instance in each category as the *reference instance* for our experiments with isomorphs. Additionally, we adopted instance in413_sp as a reference instance of special interest (see Figure 2).

max independent set
Instances fr30* are translations of a subset of unit-weighted independent set instances with hidden solution, downloaded from http://www.nlsde.buaa.edu.cn/ kexu/benchmarks/-set-benchmarks.htm. The instance dsjc125_is1 a useful test instance floating on the web, with comments that point to the original publications [16].

max clique
Instances *cliq and *cliq1 are weighted and unit-weighted instance of maximum clique problems. They have been derived from the instances fr30*, dsjc125*, and in*_sp described earlier.

blocks: min vertex cover
Instances in this set represent block compositions of increasing size of the minimum vertex cover problem. The method of block composition is described in the earlier section.

blocks: min set cover (binate)
Instances in this set represent block compositions of increasing size of the minimum binate set cover problem.

Reference instances in201_cliq, in201_cliq1, alu4, in401_sp, f51mb_0350_B_0040_20_20, have already been expanded into isomorphs; a summary of the experiments can be found in Figure 1 in the earlier section. Similarly, we expanded reference instances in401_sp, in413_sp into isomorphs; a summary of the experiments can be found in Figure 2.

In this section we re-introduce and also derive additional isomorph classes from reference instances as follows:

in401_sp_CLR, where the reference instance in401_sp.lpx represents a weighted maximum set packing problem with 500 variables.

in201_sp_CLR, where the reference instance in201_sp.lpx represents a weighted maximum set packing problem with 1000 variables.

dsjc*_CLR, where the reference instances dsjc*.lpx represent block compositions of increasing size of the minimum binate minimum vertex set problem.

f51mb*_CLR, where the reference instances f51mb*.lpx block compositions of increasing size of the minimum binate set cover problem.

It may be of some interest to observe, in Table 1, not only the column on the sparsity measure (sp) but also the column on the measure of completeness of the underlying instance graph. For example, instances in*_sc have constraint matrices that are sparse, but the underlying structure of the graph is highly 'interconnected' and hard to solve to optimality. Now, the maximum clique instances in*_cliq that have been derived from from these instances will have complement graphs that are much less 'internconnected' – and these instance have been solved to optimality in a reasonable time frame.

Experiments in Section 4 emphasized the view of *cplex* as a branch&bound solver that terminates before an externally imposed timeout. Repeating the experiments on instances from the same isomorph class allowed us to observe only one random variable, *RunTime*, since each solution represents a proven optimum which is an invariant for all instances in the class. However, note that most instances shown in Table 1 time out within 5% of the externally imposed limit of 2112 seconds – and all we have to show for it is a single value of the variable *ObjectiveBest*. Experiments that we propose for the most part of this section have been designed to produce a distribution of *ObjectiveBest* at predetermined time intervals. To get a distribution of *ObjectiveBest* on such instances, at a cost no greater than the cost of a single run with timeout value of 2112, we proceed as follows:

- take a reference instance and generate a CLR class of 32 isomorphs;
- pick a timeout value T_{out} from a set of {16, 32, 64} seconds.
- run *cplex* on the reference and all 32 instance with a timeout of T_{out} and observe the value of *ObjectiveBest* which will now become the random variable.

Table 1: Introducing a subset of reference instances and basic experiments with *cplex*.

Legend:

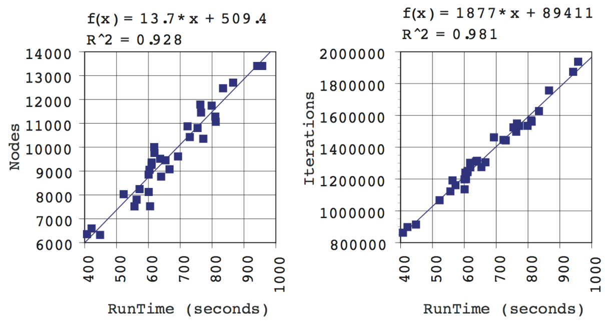
ObjBest:	values of objective function reported for each instance by cplex
Proof:	an indicator variable whether cplex has proven ‘ObjBest’ as optimal
Ones:	total number of ‘ones’ in the solution vector
RunTime:	runtime in seconds, reported by cplex
n:	number of variables
m:	number of constraints
cdMax:	maximum number of non-zero entries in a column
rdMax:	maximum number of non-zero entries in a row
sp(%):	a sparsity measure for the constraint matrix (100 * number_of_non-zeros/$(n * m)$)
gc(%):	a measure completness of the underlying graph (100 * number_of_edges/$(n * (n - 1))$ (number of unique edges is counted after expanding each constraint into a clique)

Notes:

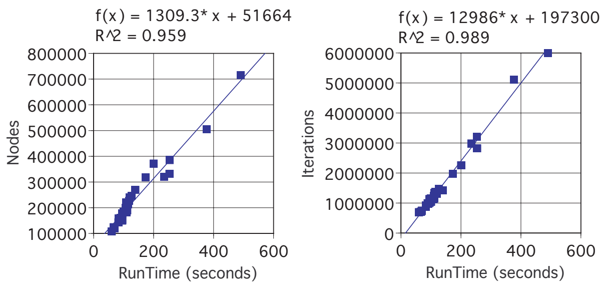
platform:	Intel-based processor, 3.2 GHz, 2 GB cache, under RedHat Linux
cplex options:	the only option used is the value of timeout (set at 2112 seconds for all instances below) (experiments with various options led to inconsistent observations)
reductions:	all matrices that represent the benchmarks in the list below have been reduced to the extent possible, using standard column and row reduction techniques [8].

Dir	Instance	ObjBest	Proof	Ones	RunTime	n	m	cdMax	rdMax	sp(%)	gc(%)
	in101_sc	189316	no	57	2112.85	1000	500	50	77	5.55	68.82
min	in201_sc	547921	no	56	2114.91	1000	1000	100	79	5.59	84.99
(unate)	in401_sc	593034	no	68	2112.52	500	1000	100	45	5.72	85.57
set	in501_sc	589992	no	54	2116.38	1500	1000	150	157	7.85	91.84
cover	in601_sc	954508	no	72	2118.01	1500	1500	150	111	5.60	90.88
	in101_sp	64408	no	19	2116.68	1000	500	50	77	5.55	68.82
max	in201_sp	77596	no	13	2117.8	1000	1000	100	79	5.59	84.99
(unate)	in401_sp	77418	yes	12	866.87	500	1000	100	45	5.72	85.57
set	in413_sp	74435	no	12	1057.95	500	1000	100	46	5.55	83.65
packing	in501_sp	76906	no	15	2118.39	1500	1000	150	157	7.85	91.84
	in601_sp	98805	no	15	2119.45	1500	1500	150	111	5.60	90.88
	dsjc125_is1	34	yes	34	17.7	125	736	23	2	1.60	9.50
max	frb30-15-1	27	no	27	2118.49	450	17827	122	2	0.44	17.65
indep.	frb30-15-2	27	no	27	2118.08	450	17874	116	2	0.44	17.69
set	frb30-15-3	28	no	28	2118.05	450	17809	122	2	0.44	17.63
	frb30-15-4	28	no	28	2118.68	450	17831	110	2	0.44	17.65
	frb30-15-5	28	no	28	2119.11	450	17794	128	2	0.44	17.61
	dsjc125_cliq1	4	yes	4	0.53	125	7014	119	2	1.60	90.50
	frb30-15-1_cliq1	15	no	15	2120.41	450	83198	407	2	0.44	82.35
max	frb30-15-2_cliq1	15	no	15	2120.02	450	83151	404	2	0.44	82.31
clique	frb30-15-3_cliq1	15	no	15	2118.98	450	83216	400	2	0.44	82.37
	frb30-15-4_cliq1	15	no	15	2118.5	450	83194	401	2	0.44	82.35
	frb30-15-5_cliq1	15	no	15	2120.63	450	83231	403	2	0.44	82.39
	in201_cliq	7265040	yes	361	3.56	1000	74959	572	2	0.20	15.01
	in201_cliq1	361	yes	361	235	1000	74959	572	2	0.20	15.01
unate	ex5.pi	36	yes	36	19.44	974	686	71	74	2.85	16.79
cover	test4.pi	105	no	105	2117.77	5117	1435	54	159	1.36	10.07
min	rot.b	84	yes	84	6.34	887	1257	158	79	1.23	7.29
binate	alu4	32	yes	32	38.5	481	592	165	74	3.46	20.16
cover	e64.b	47	no	47	2117.97	571	920	35	14	1.29	6.08
	dsjc_0125	91	yes	91	20.97	125	736	23	2	1.60	9.50
	dsjc_0250	182	no	182	2113.14	250	1472	23	2	0.80	4.73
min	dsjc_0250_0100	183	no	183	2112.98	250	1572	24	2	0.80	5.05
vertex	dsjc_0500	366	no	366	2111.15	500	2944	23	2	0.40	2.36
cover	dsjc_0500_0200	368	no	368	2112.45	500	3344	26	2	0.40	2.68
blocks	dsjc_1000	736	no	736	2126.75	1000	5888	23	2	0.20	1.18
	dsjc_1000_0400	754	no	754	2118.36	1000	7088	29	2	0.20	1.42
	dsjc_2000	1480	no	1480	2132.11	2000	11776	23	2	0.10	0.59
	dsjc_2000_0800	1511	no	1511	2116.64	2000	14976	30	2	0.10	0.75
	f51mb	12	yes	12	0.26	175	187	49	33	7.62	29.37
	f51mb_0350	24	yes	24	73.54	350	374	49	33	3.81	14.64
min	f51mb_0350_B_0040_20_20	24	yes	24	114.89	350	413	73	33	4.34	26.67
binate	f51mb_0525	36	no	36	2119.42	525	561	49	33	2.54	9.75
cover	f51mb_0525_B_0060_40_20	36	no	36	2118.11	525	660	94	53	3.45	37.82
blocks	f51mb_0700	48	no	48	2120.5	700	748	49	33	1.91	7.31
	f51mb_0700_B_0080_60_20	48	no	48	2118.25	700	925	112	73	3.11	50.13
	f51mb_1400	96	no	96	2120.55	1400	1496	49	33	0.95	3.65
	f51mb_1400_B_0160_80_80	96	no	96	2117.76	1400	2009	271	129	2.16	40.50

The isomorph class *in401_sp_CLR*



The isomorph class *f51mb_0350_B_0040_20_20_CLR*



Nodes:	the total number of nodes maintained by the branch and bound algorithm.
Iterations:	the total number of iterations done by the simplex algorithm to solve LP-relaxations at all of the nodes combined.

Figure 3: *RunTime* correlations in cplex.

Note that for value of $T_{out} = 64$, the total runtime of the experiments with (1+32) instances is 2112 seconds – however, we now may have 33 distinct values of *ObjectiveBest* in its distribution!

A summary of our experiments and observations is linked to four figures and tables that they contain.

Figure 3: We show near-perfect correlations of *RunTime* with combinatorial counts produced internally by *cplex*: *Nodes,* the total number of nodes maintained by the branch and bound algorithm, and *Iterations,* the total number of iterations done by the simplex algorithm to solve LP-relaxations at all of the nodes combined. The correlations are shown for two very different classes of isomorphs: *in401_sp_CLR* where the distribution of *RunTime* is uniform (see Figure 4), and *f51mb_0350_B_0040_20_20_CLR* where the distribution of *RunTime* is heavy-tail (see Figure 6).

Figure 4: The first three rows in the table show the statistics for *ObjectiveBest*, given the time out values of 16, 32, and 64 seconds. The fourth row shows *RunTime* statistics where an optimum value of ObjectiveBest=77418 is proven for each isomorph. The distribution is uniform, with a mean of 666 seconds, and a range from 407 to 957 seconds. The most interesting part is the fact that an optimum value of 77418 has been reached by *cplex* already in 64 seconds (by two isomorphs) – however it takes on an average of 666 seconds to prove that this value is indeed an optimum.

Figure 5: All instance reported in this figure are hard – there are no proven optima on any of the instances – despite the additional expenditures in runtime. Isomorphs in the class frb30-15-1_CLR have a known hidden solution of 30; the maximum of *ObjectiveBest* is reported at 28 – after expanding a total of 33*256 = 8448 seconds. We could not run 33 isomorphs in the class in201_sp_CLR for 2112 seconds each, the computer system timed out the experiment after solving the first 15 isomorphs.

Figure 6: The first five rows present *ObjectiveBest* statistics for 32 instances in five CLR classes: the variable size increases from 125, 250, 500, 1000, and 2000 variables and each instance is timed out at 64 seconds. These instance are compositions of blocks with hidden solution, shown in the first column of the table. Each reference instance has an number of rows (100, 200, 400, 800) that overlap the with constraints in the blocks above these rows. An optimum is proven only for the first class, one with 125 variables. No optimal solutions are found for instances beyond 125 variables.

ObjectiveBest statistics for 32 instances in five CLR classes of the binate instance f51m* are not shown. In contrast to the preceding example, *cplex* finds known optima in 16 seconds even for the largest instance (1400 variables, non-trivial number binate constraints in the overlap region). No optima can be proven for instances starting at 525 variables. However, we contrast two *RunTime* distributions for two CLR classes that can still be solved with branch&bound: f51mb_350_CLR (strictly block-diagonal, no overlap regions) and f51mb_350_B_40_20_20_CLR (non-trivial overlap of binate constraints). The statistics tabulated for two classes shows (1) a mean value of 98.4 seconds and near-exponential distribution, and (2) a mean of 232 seconds and a heavy-tail distribution. Clearly, adding overlap rows to the block composition is a significant factor in making the instance appear significantly harder (to *cplex*) – despite the fact the both instances have the same hidden solution!

7. CONCLUSIONS

This section will be completed when preparing the final version of this paper.

Acknowledgments. This work benefited a great deal from discussions, over the years, with Matt Stallmann and Xiao Yu Li. In particular, Matt Stallmann helped with the scripts that facilitated invocations of *cplex.* Eric Sills, from the NCSU High Performance Computing (HPC) facility with fast dedicated processors, assisted in a number of ways to maintain continuous access to computing resources and its environment. We also thank Y. Guo for readily sharing reprints of his papers and the 500-instance benchmark set that now has a new life in a number of settings, all in the .lpx format.

ObjectiveBest statistics for instances in isomorph classes in401_sp_CLR

(*RefV* denotes the reference instance, excluded from the computation of min, max, median, mean, and standard deviation.) Here, *branch&bound* times out at 16, 32, 64 seconds and returns the best objective value for each instance.

Class	RefV	MinV	MaxV	MedV	MeanV	StdV	N	Distribution
in401_sp_CLR@16	65086	59852	71797	65992	66080	3721	32	uniform
in401_sp_CLR@32	66826	60658	75114	69240	68548	3351	32	uniform
in401_sp_CLR@64	66826	64451	77418	70260	69829	3450	32	uniform

RunTime statistics for instances in isomorph classes in401_sp_CLR

Here, branch&bound solves each instance before time-out to an optimum value, then reports runtime.

Class	RefV	MinV	MaxV	MedV	MeanV	StdV	N	Distribution
in401_sp_CLR@BB	866	407	957	638	666	133	32	uniform

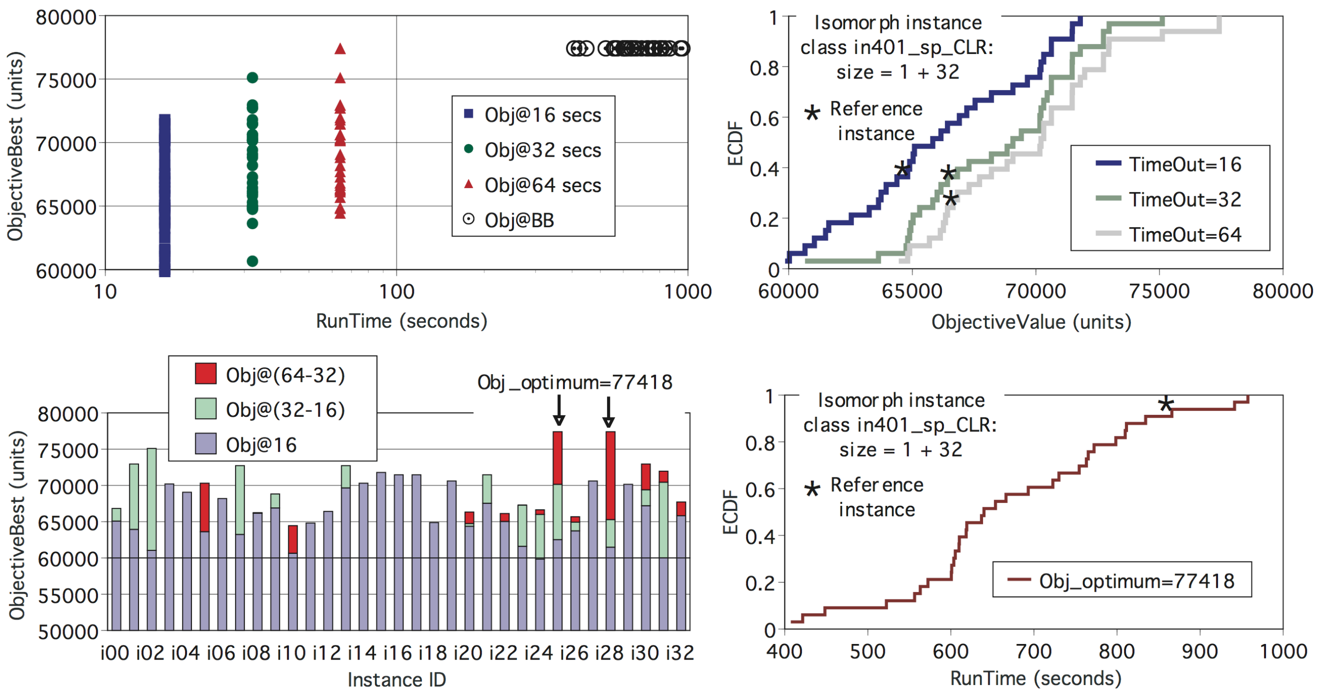


Figure 4: _Timeout_ and _branch&bound_ experiments with instances in class in401_sp_CLR.

***ObjectiveBest* statistics for instances in isomorph class frb30-15-1_CLR.**

(*RefV* denotes the reference instance, excluded from the computation of min, max, median, mean, and standard deviation.)

This is one of the independent set instances [], with hidden solution value of '30'.
The solver does not return this value, despite total computation effort of 33 * 256 = 8448 seconds –
i.e. the reference and each isomorph is run for 256 seconds before timeout.

Class	RefV	MinV	MaxV	MedV	MeanV	StdV	N	Distribution
frb30-15-1_CLR@256	26	24	28	26	26.1	0.88	32	uniform

***ObjectiveBest* statistics for instances in isomorph class in201_sp_CLR.**

Here, *branch&bound* times out at 16, 32, 64 seconds and returns the best objective value for each instance. The additional run for 2112 seconds on each isomorph does not get significantly better; it also timed-out after 15-th instance due to computer system constraints.

Class	RefV	MinV	MaxV	MedV	MeanV	StdV	N	Distribution
in201_sp_CLR@16	66948	55663	70416	60006	61034	3242	32	near-normal
in201_sp_CLR@32	66948	58656	74819	64366	64449	4146	32	normal
in201_sp_CLR@64	66948	58723	74819	66214	66510	3973	32	uniform
in201_sp_CLR@2112	77596	72417	79739	76747	76520	2243	15	uniform

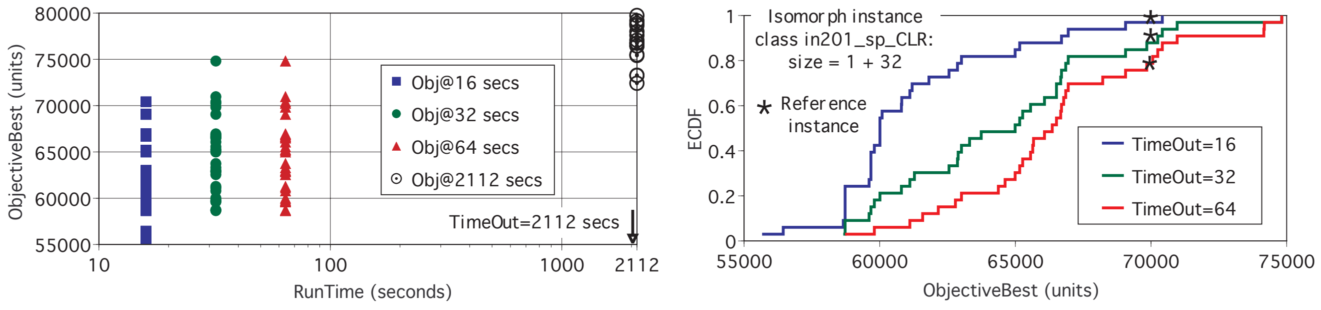


Figure 5: No optima are proven with *branch&bound* on these hard instances in class CLR.

8. REFERENCES

[1] S. Khanna, M. Sudan, L. Trevisan, and D. P. Williamson. The approximability of constraint satisfaction problems. *SIAM J. Comput.*, 30(6):1863–1920, 2000.

[2] Home page for lp_solve, 2007. http://tech.groups.yahoo.com/group/lp_solve.

[3] Home page for cplex, 2007. http://www.ilog.com/products/cplex/.

[4] X. Y. Li, M. F. M. Stallmann, and F. Brglez. Effective bounding techniques for solving unate and binate covering problems. In *DAC*, pages 385–390, 2005.

[5] F. Brglez, X. Y. Li, and M. F. M. Stallmann. On SAT instance classes and a method for reliable performance experiments with SAT solvers. *Ann. Math. Artif. Intell.*, 43(1):1–34, 2005.

[6] G. L. Nemhauser and L,A, Wolsey. *Integer and Combinatorial Optimization*. John Wiley, 1988.

[7] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Publishers, 1994.

[8] G.D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, 1996.

[9] J. E. Harlow and F. Brglez. Design of Experiments in BDD Variable Ordering: Lessons Learned. In *Proceedings of the International Conference on Computer Aided Design*. ACM, November 1998.

[10] J. E. Harlow III and F. Brglez. Design of experiments and evaluation of BDD ordering heuristics. *International Journal on Software Tools for Technology Transfer (STTT)*, 3(2):193–206, May 2001. Springer-Verlag Heidelberg. http://springerlink.metapress.com/, ISSN: 1433-2779 (Paper) 1433-2787 (Online).

[11] K. A. Brownlee. *Statistical Theory and Methodology In Science and Engineering*. Krieger Publishing, 1984. Reprinted, with revisons, from second edition, 1965.

[12] L. J. Bain and M. EngelHardt. *Introduction to Probability and Mathematical Statistics*. Duxbury, 1987.

[13] S. Yang. Logic synthesis and optimization benchmarks user guide. Technical Report 1991-IWLS-UG-Saeyang, MCNC, Research Triangle Park, NC, January 1991.

[14] Y. Guo, A. Lim, B. Rodrigues, and Y. Zhu. Heuristics for a brokering set packing problem. In *Eighth International Symposium on Artificial Intelligence and Mathematics, January 4-6, 2004, Fort Lauderdale, Florida, USA*. ACM, January 2004.

[15] Y. Guo, A. Lim, B. Rodrigues, and Y. Zhu. Heuristics for a bidding problem. *Comput. Oper. Res.*, 33(8):2179–2188, 2006.

[16] D. S. Johnson, R. Aragon C, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation; part ii, graph coloring and number partitioning. *Operations Research*, 39:378–406, 1991.

ObjectiveBest **statistics for instances in block isomorph classes, each instance times out at 64 seconds.**

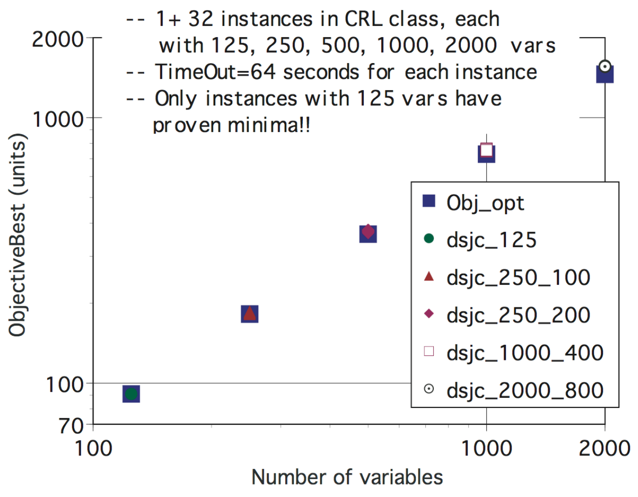
(*Opt_opt* denotes the known optimum value of the hidden solution for each block class.)
(*RefV* denotes the reference instance, excluded from the computation of min, max, median, mean, and standard deviation.)

Class	Obj_opt	RefV	MinV	MaxV	MedV	MeanV	StdV	N	Distribution
dsjc_125_CLR@64	91	91	91	91	91	91	0	32	Imp
dsjc_250_100_CLR@64	182	186	183	186	184	184	0.92	32	uniform
dsjc_500_200_CLR@64	364	372	367	378	374	374	2.81	32	uniform
dsjc_1000_400_CLR@64	728	759	747	764	756	756	4.86	32	uniform
dsjc_2000_800_CLR@64	1456	1556	1552	1569	1560	1560	4.26	32	uniform

RunTime **statistics for two block isomorph classes f51mb_0350_CLR and f51mb_0350_B_0040_20_20_CLR.**

Here, branch&bound solves each instance before time-out to an optimum value of 24, then reports runtime.

Class	Obj_opt	RefV	MinV	MaxV	MedV	MeanV	StdV	N	Distribution
f51mb_350_CLR@BB	24	73.5	50.2	446	87.8	98.4	66.7	32	near-exponential
f51mb_350_B_40_20_20_CLR@BB	24	115	71.3	2118	127	232	393	32	heavy-tail



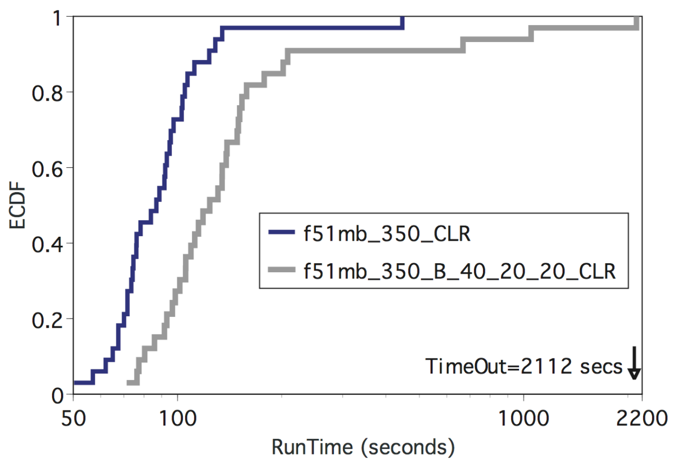


Figure 6: Asymptotic experiments with block instances generated from hidden solutions.

APPENDIX

The appendix will be completed with the revised version of this manuscript. For details and updates, interested reader may also visit

http://www.cbl.ncsu.edu/xBed/

This version of appendix includes only a brief section that illustrates the '.lpx' format. Later, we shall briefly describe software utilities used to prepare data sets (including a number of translators to/from .lpx), invoke experiments, and post-process the results.

A. SMALL EXAMPLES IN .LPX FORMAT

Lpx format appears to be an undocumented subset of the lp-file format and any pointers to its documentation will be gratefully included in the updated version of this paper. The number of hits on the web in response to a query about lpx is overwhelming and none of the listing have the context that is relevant. However, the fact remains that the two small files below will be read and produce correct results by both *lp_solve* as well as by *cplex*. We keep the emphasis on keeping the extension .lpx as a reminder that all variable names are prefixed with 'x' – a feature we rely on to post-process the respective solver outputs.

In the first file, the constraint lines are labeled explicity, a feature that is useful for a reference instance. However, as the second example shows, the constraint lines need not be labeled – a feature we find convenient when writing out an isomorph instance (in which rows are randomly permuted by design).

```
\ @file   exA_spb_max.lpx
\ @date   2007-02-01-20-26-19 (Thu Feb 01 20:26:19 GMT 2007)
\
\ ObjectiveBest   100
\ SolutionCoordinates     0110101
\ SolutionProvedOptimal   1
\
Max
  obj: +21x1  +22x2  +23x3  +25x4  +26x5  +27x6  +29x7
st
  c1:      +x2 +x3 +x4                  >= +1
  c2:      -x2         -x5 -x6          >= -2
  c3:                  +x5 +x6 -x7  >=  0
  c4:          -x3             +x7  >=  0
  c5: -x1          -x4         -x7  >= -1
  c6: -x1      -x3         -x6      >= -1
Binary
  x1  x2  x3  x4  x5  x6  x7
End
```

```
\ @file   exA_spb_max_morph_CLR.lpx
\ @date   2007-02-14-16-39-47 (Wed Feb 14 16:39:47 UTC 2007)
\ @remark below, see comments about the origin of this file
\         ------------------------------------------------
\ @VariablePermutationPairs (isomorph,reference --
\  terminated with 0,0)
\ 1,3 2,1 3,2 4,5 5,6 6,4 7,7 0,0
\
\ ObjectiveBest   100
\ SolutionCoordinates     1100011
\ SolutionProvedOptimal   1
\
Max
  obj: +22x1  +23x2  +21x3  +27x4  +25x5  +26x6  +29x7
st
  -x3 -x7 -x5 >= -1
  +x1 +x2 +x5 >= +1
  -x2 +x7 >= 0
  -x3 -x2 -x4 >= -1
  -x6 -x1 -x4 >= -2
  +x4 +x6 -x7 >= 0
Binary
  x1  x2  x3  x4  x5  x6  x7
End
```