Incorporating Simplex Method into Guided Complete Search: An Application to the Nurse Rostering Problem

Spencer K.L. Fung, Jimmy H.M. Lee, and Ho-fung Leung

Department of Computer Science and Engineering
The Chinese University of Hong Kong
Sha Tin, Hong Kong, P.R. China
{sklfung, jlee, lhf}@cse.cuhk.edu.hk

Abstract

Guided Complete Search (GCS) is a generic framework for combining and coordinating a tree search solver and a secondary solver to yield a complete and efficient CSP solver. The primary solver of GCS is systematic tree search augmented with constraint propagation algorithm, which is used to maintain the completeness of the GCS solver. The secondary solver of GCS can either be a complete or incomplete solver, which mainly produces guidance to the primary solver for solution searching by providing value ordering suggestions. In this paper, we describe our newly defined GCS/Simplex solver, which incorporates Simplex method into the GCS framework. To demonstrate the efficiency, we apply the GCS/Simplex to the nurse rostering problem, and its general form, namely, cardinality matrix problem. The nurse rostering problem is one of the most difficult scheduling problems in artificial intelligence and operations research. And the cardinality matrix problem is the underlying structure of several real-life problems such as rostering, scheduling and time-tabling problem. Both are shown to be computational intractable. Experimental results show that the GCS/Simplex solver is efficient in solving this kind of scheduling problems in terms of both computation time and number of fails.

Introduction

Solver collaboration is crucial in constraint satisfaction problem (CSP) research. Recently, it draws attention in combining different solvers yields a more efficient hybrid solver for CSPs. Many examples have been reported in the literature. (Lee et al. 1996) propose that derivations in constraint logic programming could be guided by GENET (Wang and Tsang 1991), a stochastic search based solver for binary CSPs. Their approach consists of employing a GENET solver at each choice point. Before the sub-trees rooted at the choice point are explored, the GENET solver is invoked to solve to provide guidance to search. (Hooker et al. 1999) proposed mixed logic/linear programming (MLLP) that makes use of conditionals to link discrete and continuous elements of the problem, and brings the idea of integration of a checker with a solver. (Gomes et al. 2002) proposed randomized backtrack search which employs linear programming. In our previous work, Guided Complete Search (GCS) has been formulated for solving hard CSP instances (Fung et al. 2004). GCS is a generic hybrid scheme, which parameterized by a secondary solver. In GCS, guidance for the tree search based solver in the form of value ordering suggestions are generated automatically by the secondary solver, without specific prior design or domain knowledge. The main idea of GCS is that the operations of a tree search based solver are coordinated with those of another solver, which maintain the soundness and completeness of the whole hybrid scheme. The value commitments made by the tree search based solver and the results of constraint propagation (Mackworth 1977) help the secondary solver to reduce the problem size in order to speed up the whole search process.

In this paper, we instantiate the GCS with the Simplex method (Wolsey 1998), obtaining a new GCS/Simplex hybrid solver. The Simplex method is an efficient iterative algorithm to solve linear programming problems (LP), which achieves low average case complexity. The problem domain handled by Simplex is in the real domain, and the constraints are linear. Therefore, we need to transform the original CSP into a LP for the Simplex in order to enable collaboration between these two different types of solvers. The results of constraint propagation greatly reduce the computation overhead in solving the linear relaxation of CSP, while the fractional solution generated by Simplex can guide the tree search towards a complete solution. GCS/Simplex is a general CSP solver, it able to solve certain hard problem instances. To demonstrate the efficiency of this solvers combination, we apply the GCS/Simplex solver to the nurse rostering problem (Cheng et al. 1997) and its general form, namely, cardinality matrix problem (Regin and Gomes 2004). The nurse rostering problem is one of the most difficult scheduling problems in artificial intelligence and operations research. In general, it consists of cardinality constraints and some of the user-defined constraints for prohibiting a specific shift pattern to appear. The search space of the problem is usually large and the problem structure is complex. Hence the problem becomes extremely difficult to solve by a general CSP solver when domain-specific heuristics is not given. The problem is computational intractable. Various approaches have been proposed to solve the problem. (Aickelin and White 2004) employ evolutionary algorithm approaches, which is sensitive to the parameters setting and the chose of crossover operator. Therefore, performance can be greatly affected by the combination of parameters. (Wong and Chun to appear) propose a problem-specific approach for solving this particular problem, which is a tree search based CSP solver incorporating meta-level reasoning and probability-based ordering heuristics with JSolver (Chun 1999). (Thornton and Sattar 1997) propose a special integer programming (IP) model, which combines two IP models, and introduces problem decomposition heuristic to compute a feasible solution for the problem. Besides the nurse rostering problem, we describe benchmarking results on solving cardinality matrix problem, which is a generalization of nurse rostering problem. It is also the underlying structure of several real-life problems such as rostering, scheduling, time-tabling and so on. These are hard computational problems given their inherent combinatorial structure. The GCS/Simplex solver has been shown efficient in solving this kind of combinatorial problem in terms of computation time and number of fails, which outperform a classical CSP solver and a mixed integer programming (MIP) solver.

The paper is organized as follows. We describe the background of CSP in the next section. And we present the Guided Complete Search (GCS) and the proposed GCS/Simplex solver in the third section. In the fourth section, we describe the nurse rostering problem as well as the cardinality matrix problem, and report the experimental results. We give a conclusion in the last section.

Constraint Satisfaction Problem

A *Constraint satisfaction problem* is a tuple $\langle X, D, C \rangle$ (Mackworth 1977), where $X = \{x_1, \ldots, x_n\}$ is a finite set of *n* variables, $D = \{d_1, \ldots, d_n\}$ is a set of variable domains, and $C = \{c_1, \ldots, c_m\}$ is a finite set of *m* constraints over the variables in *X*. Each domain $d_i \in D$, $1 \le i \le n$, is a finite and discrete set of constants, and each constraint $c_i \in C$, $1 \le i \le m$, is a relation over a finite subset of *X*. An assignment $A = \langle x_1 \mapsto a_1, \ldots, x_n \mapsto a_n \rangle$ is an *n*-tuple where $a_i \in d_i$, $1 \le i \le n$. An assignment *A* is a solution to a constraint satisfaction problem if and only if all constraints in *C* are satisfied by *A* after replacing every occurrence of variable x_i by a_i. It is required that the satisfiability of each constraint be decidable. To find a solution to a CSP is a well-known *NP*-complete problem.

A common approach for solving CSP is to employ a tree search method. Many commercially available CSP solvers, such as CHIP (Dincbas et al. 1988), ILOG Solver (ILOG 2003) and JSolver (Chun 1999), are based on this approach. The search is performed in depth-first search from left to right fashion. Failure is detected at each node (or *choice point*) of the search tree. In the case of a failure, the search will continue at the next available node. We call such process as a backtracking. Many techniques have been proposed to improve the efficiency of tree search approaches, which can be classified into three main categories. First, *variable-ordering heuristics* aim to select appropriate variable to be instantiate at each choice point, so that the size of the search tree might be reduced by earlier failures. On the other hand, *value-ordering heuristics* aim at identifying and trying the most promising values first at each choice point, so that a first solution can hopefully be found earlier. Finally, *constraint propagation*, such as node- and arc-consistency algorithms (Mackworth 1977), can be employed at each choice point to reduce the size of the variable domains, and thus also the size of the search tree as some of the branches are pruned in the process. Among these techniques, applying value-ordering heuristics is common and important to speed up the searching process, which can guide the search to reach a first solution opportunistically. However, it is difficult to know what value-ordering heuristics are suitable for different problems, as the effectiveness of value-ordering heuristics is problem-dependent. One of the ideas of the GCS framework is to automate value-ordering procedure.

Guided Complete Search

In a nutshell, GCS is a hybrid scheme for combining two solvers, a primary and a secondary solver. It coordinates the collaboration of the primary complete tree search-based solver (TS) and a secondary solver in order to produce a complete and efficient CSP solver. Each solver approaches solutions in different way. Therefore, insightful information can be discovered in different aspects as that the two solvers can help each other. From the viewpoint of TS, the secondary solver acts as an “oracle” that generates heuristics for value ordering. Meanwhile, TS narrows the search space for the secondary solver by constraint propagation after each value commitment. The information exchange operations improve the performance for both solvers in the framework. As a result, a more efficient hybrid solver can be obtained. We can instantiate the GCS scheme by incorporating different solver as a secondary solver. We gave GCS/X to denote that the GCS instance incorporates a solver X.

The Framework

A general framework of GCS is shown in Figure 1. The interface function the_secondary_solver_returns connects the two solvers, which either returns a solution *A*, a *failure* (“$\perp$”), or an *unknown*. The return of *unknown* in the secondary solver usually refers to the case of reaching a pre-set resource limit. In such a case the secondary solver should demonstrate a favor of values of some variables. Hence, val_suggestion_from_the_secondary_solver is called, and then the variable labeling procedure proceeds in TS. Eventually, a solution can be found either by the TS solver or by the secondary solver. The cost of employing the secondary solver to obtain a promising value for a variable might be expensive. A cost-reducing version of GCS is described in our previous work (Fung et al. 2004), which provides flexibility for different degree of integration. One example is that, the tree search seeks for value suggestion when deep backtrack performs.

```
GCS( $X, D, C$ ) {
  $\langle X, D', C \rangle$ = constraint_propagation( $\langle X, D, C \rangle$ );
  if ( $\exists d[d \in D' \rightarrow d = \varnothing]$ ) return $\perp$;
  if ( $\forall d[d \in D' \rightarrow |d| = 1]$ )
    return $\{x_i \mapsto s_i \mid i = 1, 2, \ldots, n\}$, where $d_i = \{s_i\}$;
  update_the_secondary_solver( $\langle X, D', C \rangle$ );
  R = the_secondary_solver_returns();
  if (R = $A$) return $A$;
  if (R = $\perp$ ) return $\perp$;
  $i$ = var_ordering_heuristics ($X$);
  repeat {
    $a$ = val_suggestion_from_the_secondary_solver($i$);
    update_the_secondary_solver( $\langle X, D', C \cup \{x_i = a\} \rangle$ );
    if (the_secondary_solver_returns() = $A$) return $A$;
    if (not the_secondary_solver_returns() = $\perp$ ) {
      $A$ = GCS( $\langle X, D', C \cup \{x_i = a\} \rangle$ );
      if ( $A \neq \perp$ ) return $A$;
    }
    $d_i = d_i \setminus \{a\}$;
  } until $d_i = \varnothing$;
  return $\perp$;
}
```
Figure 1. A framework of GCS solvers for CSPs

The search tree built from the GCS framework is basically a search tree of canonical complete tree search. The only difference locates at the variable labeling procedure, which is guided by the results returned by the function the_secondary_solver_returns. The secondary solver acts as the value ordering heuristics function to determine which value to instantiate next. Provided the function the_secondary_solver_returns always returns within a finite period of time, the GCS framework is sound and complete.

The Secondary Solver: Simplex method

The GCS/Simplex employs Simplex method (Wolsey 1998) as secondary solver in GCS. The Simplex method is an iterative procedure for LP. It solves a system of linear equations in each of its steps and terminates when either the optimum or solution infeasibility is reached. Intrinsically, the Simplex method solves optimization problems. However, for solving CSPs, objective function will not be presented. In addition, linear relaxation is needed in order to apply Simplex method for integer problem, which is a common approach in integer programming (IP). Similarly, a linear relaxation model of the original CSP is required in GCS/Simplex. We transform a CSP into a 0-1 LP, which the set of 0-1 variables $b \in B$ in the LP are bounded by $0 \leq b \leq 1$. In a nutshell, the problem transformation is to map each value in a variable domain of the CSP into a corresponding 0-1 variable in the LP. For example, a variable x with domain $\{0,1,2\}$ in CSP, there is a set of corresponding 0-1 variables $\{b_0, b_1, b_2\}$ in LP. And, each constraint in CSP is transformed into linear form in LP.

Constraint Linearization

After creating a set of 0-1 variables B in the LP model, the key point is how to transform a set of constraints in CSP into a set of linear constraints. We present three types of widely used constraint in this section, including *illegal*, *alldifferent*, and *cardinality* constraint. Besides, we need to model the intrinsic requirement for each variable in CSP, in which each variable can only take one value at a time, as a linear constraint. We represent the requirement in LP with the equation, $\sum_{\forall j \in d_i} b_{ij} = 1$, for $i = 1, 2, ..., n$. This so-called *single-value* constraint is essential in the GCS/Simplex framework for each problem.

The *Illegal* constraint (Wang and Tsang 1991) is a generic constraint in CSP. It is specified by a set of incompatible variables assignments. Suppose a constraint c involves m variables $\{x_1, ..., x_m\}$, with an incompatible tuple $(a_1, ..., a_m)$, we add one linear inequity $\sum_{i=1}^{m} b_{ia_i} \leq m-1$, where $a_i \in d_i$, to the LP model. It achieves the same effect as the original constraint by prohibiting the participating variables from taking corresponding values in an incompatible tuple simultaneously. In general, any type of constraints can be modeled as illegal constraint, but the resulting model becomes too bulky for representing all incompatible tuples in a problem. Therefore, it is used for modeling complicated constraint, such as pattern constraint that is discussed in the next section.

The *alldifferent* constraint (Hoeve 2001) is present in most commercial constraint programming systems. It has been widely applied in many real-life problems, due to the special constraint semantic and the desirable pruning power. The constraint states that all involving variables must be pair-wisely different, which defines *alldifferent*$(x_1, ..., x_m)$ = $\{(a_1, ..., a_m) \mid a_i \in d_i, a_i \neq a_j \text{ for } i \neq j\}$. We express the constraint as, $\sum_{i=1}^{m} b_{ij} \leq 1$ for all $j \in d_i$.

The *cardinality* constraint (Hentenryck rt al. 1992) is defined over a set of variables $\{x_1, ..., x_m\}$, and specified by two integer values n and v. It states that the number of variables instantiating to a value v must be within a given range specified by n. The *atmost*, *atleast*, and *exactly* constraint are instances of the cardinality constraint. For example, the $atmost(n, \{x_1, ..., x_m\}, v)$ constraint specifies that at most n variables in $\{x_1, ..., x_m\}$ is allowed to take the value v. This type of constraint commonly occurs in rostering, scheduling and time-tabling problems to restrict the allocating resources. We represent the *atmost*, *atleast*, and *exactly* constraint with $\sum_{i=1}^{m} b_{iv} \leq n$, $\sum_{i=1}^{m} b_{iv} \geq n$, $\sum_{i=1}^{m} b_{iv} = n$ respectively. Recently, a generalization of cardinality constraint is presented (Regin 1996), namely, *Global Cardinality Constraint* (GCC), which handles a bunch of cardinality constraints.

Interaction between GCS and Simplex

In order to connect the two models, we have to maintain the following constraints during the search process, for $i = 1, 2, ..., n$ and all $j \in d_i$:

$$\begin{aligned} x_i = j \Leftrightarrow b_{ij} = 1 \\ x_i \neq j \Leftrightarrow b_{ij} = 0 \end{aligned} \qquad (3.1)$$

where $x_i \in X$ is a variable of the CSP, and $b_{ij} \in B$ is a variable of the LP. According to (3.1), an assignment $x_i \mapsto j$ can be made when the corresponding 0-1 variable b_{ij} is bounded to "1", or b_{ij} is strictly greater than 0. Similarly, if a domain value j of x_i is removed, the corresponding variable b_{ij} is set to "0". A domain value j of x_i can also be removed, if b_{ij} is strictly less than 1. The Simplex method is applied to the linear relaxation of the original CSP, when the function the_secondary_solver_returns is invoked, to compute a solution. If the solution is integral, then it returns the complete solution. Otherwise, it suggests a value j to TS for a particular variable x_i, which the corresponding 0-1 variable b_{ij} has a fractional value, when the function val_suggestion_from_the_secondary_solver is invoked.

An Illustrative Example

We use an *N-Queen* problem to illustrate how the GCS/Simplex works, and also to demonstrate the guiding power by the Simplex method. A comparison between a tree search based CSP solver and GCS/Simplex has been made in terms of number of fails, which is a common measure for evaluating the efficiency of value ordering heuristic. The less fails occurs, the more accurate guidance.

An *N-Queen* is the problem of placing n queens on a $n \times n$ chessboard so that no queens can take on one another. A queen attacks another queen when both of them are placed on the same row, column or main diagonals. (Hoeve 2001) suggests a CSP model for this problem by using the alldifferent constraint, in which a variable x_i represents column $i = 1, 2, \ldots, n$, which ranges over row 1 to n. This means that, in every column i, a queen is placed in the x_i-th row. The constraints are expressed as follows,

$$alldifferent(x_1, ..., x_n) \qquad (3.2)$$

$$alldifferent(x_1 - 1, x_2 - 2, ..., x_n - n) \qquad (3.3)$$

$$alldifferent(x_1 + 1, x_2 + 2, ..., x_n + n) \qquad (3.4)$$

$$x_i \in \{1, 2, ..., n\} \text{ for } 1 \leq i \leq n$$

And, the LP model can be obtained by the aforementioned alldifferent constraint linearization and the single-value constraint, which shows as following,

$$\sum_{j=1}^{n} b_{ij} = 1 \quad \text{for } i = 1, 2, ...n \qquad (3.5)$$

$$\sum_{i=1}^{n} b_{ij} \leq 1 \quad \text{for } j = 1, 2, ..., n \qquad (3.6)$$

$$\sum_{b \in Y_i^+} b \leq 1 \quad \text{for } i = 1, .., 2n-3, \ Y_i^+ \subseteq B \qquad (3.7)$$

$$\sum_{b \in Y_i^-} b \leq 1 \quad \text{for } i = 1, .., 2n-3, \ Y_i^- \subseteq B \qquad (3.8)$$

$$b_{ij} \in B, \text{ for } i, j = 1, 2, ...n$$

where Y_i^+ and Y_i^- is a set of 0-1 variables that rely on the i-th positive and negative main diagonal of the chessboard respectively. The constraint (3.5) is the single-value constraint; (3.2) and (3.6) represent no queens are allowed to occur in the same row; (3.3) and (3.7), and (3.4) and (3.8) represent no queens can be put on the same positive and negative main diagonals of the chessboard respectively.

We have applied ILOG solver 6.0 (ILOG 2003) and GCS/Simplex solver on a set of *N-Queen* problem instances ranging from $n = 4$ to 100. The GCS/Simplex is able to solve all instances in the experiment. The results of the number of fails are shown in Figure 2. It is obvious that the GCS/Simplex is superior to ILOG solver in terms of number of fails. In many cases, the GCS/Simplex achieve zero fails that means no backtrack occurs during the search, which also indicates the search is guided to a solution accurately. Similar results can be obtained from another famous CSP benchmark, such as quasi-group completion problem (Gomes and Shmoys 2002). A more comprehensive evaluation is discussed in the next section.

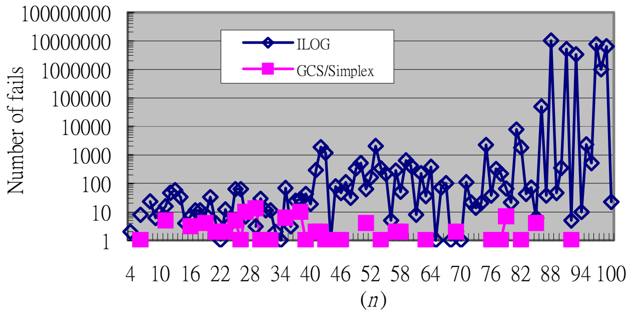

Figure 2. ILOG Solver vs. GCS/Simplex on *N-Queen*

Nurse Rostering Problem

Nurse Rostering is a task of deciding when a nurse should report to work each day in order to fulfill the predicted workforce demand in a hospital ward. This demand may vary from shift to shift and from day to day. Besides fulfilling the demand, a rostering system must also ensure that each nurse is assigned enough work per week and that each nurse should get adequate rest between shifts. This specific issue was discussed in the past studies (Cheng et al. 1997). The specifications of the problem vary from hospitals and the user requirements, but the problem structure is similar. A typical nurse rostering problem specification can also be found in (Cheng et al. 1997), which is a case study in the Tang Shiu Kin Hospital of Hong Kong. The nurse rostering problem is well-known difficult in artificial intelligence and operations research.

Problem Descriptions

For a general nurse rostering problem, four essential shift types are taken into consideration: morning-shift (A), afternoon-shift (P), night-shift (N), and day-off (DO). Note that some special type of shifts might be available in practice; refer to (Cheng et al. 1997). The constraints for making a working roster can be divided into three categories: *daily constraint*, *weekly constraint*, and *shift pattern constraint*. A working roster example is shown in Figure 3. It consists of three types of table. The upper-left one shows the shift for each nurse for one week. The bottom table is a tally of the total number of nurses working different shifts each day. This table is used to verify if the daily work demand is sufficiently met. The right-hand-side table is a tally of the total number of shifts worked by each nurse for one week. This table is used to check if enough work is assigned to each nurse.

Name	Sun	Mon	Tue	Wed	Thu	Fri	Sat
John	N	DO	A	A	A	P	A
Kate	A	N	DO	P	P	A	A
Tom	DO	A	P	A	A	A	N
Susan	A	A	N	N	DO	P	A
Jan	P	P	A	P	A	N	DO
Nancy	P	P	A	P	N	DO	A
Linda	DO	A	A	P	A	N	DO
David	DO	A	P	A	P	A	N
Jerry	P	A	P	A	N	DO	P
Amy	N	DO	A	A	P	A	P
Mary	DO	P	N	N	DO	P	P
Bill	A	N	DO	A	A	A	DO

A	P	N	DO
4	1	1	1
3	2	1	1
4	1	1	1
3	1	2	1
2	3	1	1
2	3	1	1
3	1	1	2
3	2	1	1
2	3	1	1
3	2	1	1
0	3	2	2
4	0	1	1

A	3	5	5	6	5	5	4
P	3	3	3	4	3	3	3
N	2	2	2	2	2	2	2

Figure 3. A working roster example

A *daily constraint* defines the demand for nurses per shift per day, which is depicted in the bottom table of Figure 3. This constraint can be modelled by an *exactly* constraint that described in the previous section. For example, on Sunday, there should be exactly 3 nurses in the morning and afternoon, and 2 nurses at night. On Monday, Tuesday, Thursday, and Friday, there should be exactly 5 nurses in the morning, 3 in the afternoon, and 2 during night. On Wednesday, there should be exactly 6 nurses in the morning, 4 in the afternoon, and 2 during overnight. Saturday is like Sunday, but with 1 additional nurse in the morning. A *weekly constraint* defines the expected work from each nurse within the week, which is depicted in the right-hand-side table of Figure 3. The following is an example of weekly constraints for the nurse, John. He can work at most 4 morning shifts per week. He should work at least 1 afternoon shift per week. He should work at least 1 night shift per week. He should get at least 1 day of rest per week. A *shift pattern constraint* defines desirable shift assignment sequences for a nurse. Typically, a nurse after working a night shift who should be given a day off, or should continue to work on another night shift. Therefore, the constraint prohibits the shift pattern N-A or N-P to be assigned for a nurse for two consecutive days.

CSP Model

We model the nurse rostering problem as a CSP, which is based on the format of a working roster shown in Figure 3, in the viewpoint of allocating shifts to nurses. A variable represents a shift on a day-of-week for a nurse with a domain $\{0,...,3\}$, which represents shift type DO, A, P, and N respectively. Each row in the roster consists of 7 variables, which holds the shifts assigned to a nurse in a scheduled week, i.e. there are totally 84 variables are created for a twelve-nurse hospital ward. The daily and weekly constraint can be simply modeled by cardinality constraint, while the shift pattern constraint can be modeled by illegal constraint in the CSP model. The corresponding linear relaxation model can be transformed from the CSP model in the same way as show in the N-Queen example.

Experimental Results

A GCS/Simplex solver is implemented with an ILOG Solver 6.0 (ILOG 2003) and a COIN-OR LP solver (CLP). Experiments are conducted on the selected test cases from (Wong and Chun to appear), which are shown extremely difficult to solve. A comparison between ILOG Solver, GCS/Simplex, and a COIN-OR Branch-and-Cut solver (CBC) is shown in Table 1. Note that the Simplex engine in CBC is the same as the one in GCS/Simplex, and CLP and CBC are available online at http://www.coin-or.org. The solvers have been executed on a Sun Enterprise E4500 UNIX workstation, and the execution time limit is 2 hours.

The GCS/Simplex solver is superior to both ILOG solver and the MIP solver in terms of computation time and number of fails. ILOG Solver cannot find a solution or proof the problem unsatisfiability within the time limit in this experiment; while the GCS/Simplex solver solves all problem instances less than a second, which outperforms the MIP solver by an order of magnitude.

	Runtime (sec)			Number of fails		
	ILOG	GCS/Simplex	CBC	ILOG	GCS/Simplex	CBC
Test Case 3	> 7200	**0.53**	9.92	---	**0**	n/a
Test Case 4	> 7200	**0.72**	7.9	---	**0**	n/a
Test Case 7	> 7200	**< 0.01**	0.12	---	**0**	n/a

Table 1. Results on nurse rostering problems

The Cardinality Matrix Problem

To further demonstrate the efficiency of GCS/Simplex, we use the cardinality matrix problem introduced in (Regin and Gomes 2004). It is the underlying structure of several real-life problems such as rostering, scheduling, and time-tabling problem. The problem consists of a $m \times n$ matrix of integer variables X_{ij} for $1 \le i \le m$ and $1 \le j \le n$. There is a global cardinality constraint on each row and column of the matrix to restrict the occurrences of a value in a set of variables. The problem is computational intractable. We propose a $n \times n$ cardinality matrix problem as a benchmark in this experiment. The domain of each variables ranging from 0 to $\lfloor n/2 \rfloor$. The problem requires each value in the domain has to be appeared at least once but no more than twice on each row and column.

The results are shown in Table 2. It is obvious that the hard instances appear from $n = 12$ onwards. The ILOG solver solves the instances from $n = 2$ to 11 efficiently. However, it shows difficult in solving those large instances for $n \geq 14$. The GCS/Simplex solver solves all instances in a reasonable time. Besides, the results of number of fails are noticeable, no backtrack occurs during the GCS/Simplex search. It indicates that the Simplex method guides the search effectively.

	Runtime (sec)			Number of fails		
	ILOG	GCS/Simplex	CBC	ILOG	GCS/Simplex	CBC
$N = 2$	**< 0.01**	**< 0.01**	0.02	**0**	**0**	n/a
$N = 3$	**< 0.01**	**< 0.01**	0.01	**0**	**0**	n/a
$N = 4$	**0.01**	0.02	0.03	**0**	**0**	n/a
$N = 5$	**0.01**	0.03	0.04	**0**	**0**	n/a
$N = 6$	**0.01**	0.11	0.17	**0**	**0**	n/a
$N = 7$	**0.02**	0.26	1.67	**0**	**0**	n/a
$N = 8$	**0.03**	0.53	15.94	3	**0**	n/a
$N = 9$	**0.01**	1.1	22.25	6	**0**	n/a
$N = 10$	**0.03**	2.98	118.1	9	**0**	n/a
$N = 11$	**0.03**	4.47	193.63	1	**0**	n/a
$N = 12$	27.04	**12.67**	764.76	89956	**0**	n/a
$N = 13$	**0.09**	14.45	1465.45	33	**0**	n/a
$N = 14$	> 7200	**48.63**	3078.07	---	**0**	n/a
$N = 15$	**11.19**	62.49	4295.56	28440	**0**	n/a
$N = 16$	> 7200	**140.33**	7028.04	---	**0**	n/a
$N = 17$	**163.95**	166.75	> 7200	371837	**0**	n/a
$N = 18$	> 7200	**356.65**	> 7200	---	**0**	n/a
$N = 19$	> 7200	**442.77**	> 7200	---	**0**	n/a
$N = 20$	> 7200	**813.01**	> 7200	---	**0**	n/a
$N = 21$	> 7200	**1085.93**	> 7200	---	**0**	n/a
$N = 22$	> 7200	**1980.1**	> 7200	---	**0**	n/a
$N = 23$	> 7200	**2416.65**	> 7200	---	**0**	n/a
$N = 24$	> 7200	**4260.76**	> 7200	---	**0**	n/a
$N = 25$	> 7200	**5509.57**	> 7200	---	**0**	n/a

Table 2. Results on the cardinality matrix problem

Concluding Remarks

This paper presents a newly defined GCS/Simplex solver, which incorporates Simplex method into the guided complete search framework. In order to illustrate the efficiency of the GCS/Simplex solver for solving hard CSP instances, we apply the GCS/Simplex to nurse rostering problem and cardinality matrix problem. It is one of the hardest problems in artificial intelligence and operations research. In most of the cases in the experiment, the GCS/Simplex achieve zero backtrack that indicates the guidance provided by the Simplex method is promising and directs the search toward a solution. Experimental evidence shows that GCS/Simplex is able to solve certain hard problems without specific prior design or domain knowledge, which outperforms a tree search based CSP solver and a mixed integer programming solver.

Acknowledgements

The work described in this paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK4211/01E).

References

Aickelin, U. and White, P. 2004. Building Better Nurse Scheduling Algorithms. *Annals of Operations Research* 128:159-177.

Cheng, B.M.W., Lee, J.H.M. and Wu J.C.K. 1997. A Nurse Rostering System Using Constraint Programming and Redundant Modeling. *IEEE Transactions on Information Technology in Biomedicine* 1:44-54.

Chun, H.W. 1999. Constraint Programming in Java with JSolver. In *Proceedings of the First International Conference and Exhibition on The Practical Application of Constraint Technologies and Logic Programming*.

Dincbas, M., Hentenryck, P. V., Simonis, H., Aggoun, A., Graf, T., and Berthier, F. 1988. The Constraint Logic Programming Language CHIP. In *Proceedings of the Fifth Generation Computer Systems*, 693-702.

Fung, S.K.L., Zheng, D.J., Leung, H.F., Lee, J.H.M. and Chun, H.W. 2004. A Framework for Guided Complete Search for Solving Constraint Satisfaction Problems and Some of Its Instances. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence*.

Games, C.P. and Shmoys, D.2002. The promise of LP to boost CSP techniques for combinatorial problems. In *Proceedings of the Fourth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems*.

Gomes, Carla P. and Shmoys, D. 2002. Completing quasigroups or latin squares: A structured graph coloring problem. In *Proceedings of the Computational Symposium on Graph Coloring and Extensions*.

Hentenryck Van Pascal, Simonis H. and Dincbas M. 1992. Constraint Satisfaction using Constraint Logic Programming. *Artificial Intelligence* 58:113-159.

Hoeve, van W.J. 2001. The Alldifferent Constraint: A Survey. In *Proceedings of Sixth Annual Workshop of the ERCIM Working Group on Constraints*.

Hooker, J.N. and Osorio, M.A. 1999. Mixed logical-linear programming. *Discrete Applied Mathematics* 96-97(1-3):395-442.

ILOG Inc., S. A., Gentilly, France. 2003. ILOG Solver 6.0, *User Manual*.

Lee, J.H.M., Leung, H.F., Stuckey, P.J., Tam, V.W.L., and Won, H.W. 1996. Using Stochastic Methods to Guide Search in CLP: a Preliminary Report. In *Proceedings of Asian Computing Science Conference*, 43-52.

Mackworth, A. 1977. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99-118.

Regin, Jean-Charles 1996. Generalized arc consistency for global cardinality constraint. In *Proceedings of the 13th National Conference on Artificial Intelligence*.

Regin, Jean-Charles and Gomes, Carla P. 2004. The Cardinality Matrix Constraint. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, 572-587.

Thornton, J. R., and Sattar, A. 1997. Nurse Rostering and Integer Programming Revisited. In *Proceedings of International Conference on Computational Intelligence and Multimedia Applications*, 49-58.

Wang, C.J. and Tsang, E.P.K. 1991. Solving constraint satisfaction problems using neural-networks. In *Proceedings of the IEE Second International Conference on Artificial Neural Networks*, 295-299.

Wolsey, L.A. 1998. *Integer programming.*: John Wiley.

Wong, G.Y.C. and Chun, H.W. to appear. Constraint-based rostering using meta-level reasoning and probability-based ordering. *Engineering Applications of Artificial Intelligence*.

Regin, Jean-Charles 1996. Generalized arc consistency for global cardinality constraint. In *Proceedings of the 13th National Conference on Artificial Intelligence*.