A Framework for Guided Complete Search for Solving Constraint Satisfaction Problems and Some of Its Instances

Spencer K.L. Fung,[†] Denny J. Zheng,[†] Ho-fung Leung,[†] Jimmy H.M. Lee,[†] and H.W. Chun[‡]

[†]*Dept. of Computer Science and Engineering*
The Chinese University of Hong Kong
Sha Tin, Hong Kong, P.R. China
{sklfung, jyzheng, lhf, jlee}@cse.cuhk.edu.hk

[‡]*Department of Computer Science*
City University of Hong Kong
Kowloon, Hong Kong, P.R. China
andy.chun@cityu.edu.hk

Abstract

Systematic tree search augmented with constraint propagation has been regarded as the de facto standard approach to solve constraint satisfaction problems (CSPs). The property of completeness of tree search is superior to incomplete stochastic local search, even though local search approach is more efficient in general. Many heuristics techniques have been developed to improve the efficiency of the tree search approach. In this paper, we propose a framework for combining and coordinating a complete tree search solver and a different solver in order to produce a complete and efficient CSP solver. Four different instances of the framework have been suggested, which include combining complete tree search with stochastic search, mathematical programming approach respectively. The experimental results show that this highly integrated hybrid scheme greatly improve the efficiency of constraint solving process in terms of both computation time and number of backtracking.

1. Introduction

Research into Constraint Satisfaction Problems (CSPs) [14] is an active research area in the realm of artificial intelligence. CSPs find numerous practical applications in the real life, including resource allocation [4], rostering [1], scheduling, timetabling [3], network design [5] and so on. Different solvers for CSPs have been proposed, which can be roughly divided into two main categories. On the one hand, tree search based solvers systematically explore virtually the whole search space. These solvers, though relatively less efficient, are complete in the sense that all solutions can eventually be found. On the other hand, stochastic solvers, such as artificial neural networks (ANNs) [18], evolutionary algorithms (EAs) [16] and Genetic Algorithms (GA) [17], are usually practically faster by orders of magnitude, but they do not guarantee to find any solutions even if solutions exist.

Heuristics is commonly employed to improve the efficiency of CSP solvers. For examples, value ordering and variable ordering heuristics can often significantly improve the efficiency of tree search based solvers. Occasionally, domain specific heuristics, such as the Least-Loaded Routing (LLR) strategy [5] for maximizing the network throughput, are sometimes available..

We observe that some solvers are able to demonstrate obvious favor of particular values of some variables in the process of searching for solutions, even before any solution is found. These favored values often turn out to constitute (part of) the solutions eventually found. For example, when applying Genetic Algorithms (GAs) to solve CSPs, it is almost always the case that most members of the surviving population favor particular values of the variables.

In this paper, we propose a hybrid scheme for solving CSPs. In the proposed scheme, the operations of a tree search based solver are coordinated with those of another solver that demonstrate favor for particular value(s) of variables in the process of CSP solving. The favored values are made use of by the tree search based solver as value ordering heuristics. Meanwhile, value commitments made by the tree search based solver, as well as other information such as the results of constraint propagation, help the collaborating solver to reduce the problem size. The hybrid solver as a whole is a complete solver. Experiments show that, with a suitable choice of the collaborating solver, this highly integrated hybrid scheme greatly improve the efficiency of constraint solving process.

Solver collaboration in solving CSPs has been reported in the literature. In 1996 Lee *et al.* [13] proposed that the derivation in constraint logic programming (CLP) could be guided by GENET [18], a stochastic search based solver for binary constraint satisfaction problems. In a nutshell, 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. Hooker *et al.* [8,9,10] proposed 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.* [6,7] proposed randomized backtrack search which employs linear programming.

The paper is organized as follows. In the next section, we describe the background of the CSPs. The Guided Complete Search (GCS) framework is presented in section 3. After that, various GCS schemes and experimental results are presented. Finally, we conclude the paper and give the direction our future work.

2. Background

The definition of constraint satisfaction problems (CSP) and the incorporated search algorithm will be described in this section.

2.1. Constraint Satisfaction Problems

A *constraint satisfaction problem* [14] is a three-tuple $\langle X, D, C\rangle$, where $X = \{x_1, \ldots, x_n\}$ is a finite set of *n* variables, $D = \{d_1, \ldots, d_n\}$ where d_i is the *domain* of variable x_i, and $C = \{c_1, \ldots, c_m\}$ 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* of variables is an *n*-tuple $A = \langle x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\rangle$ 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 the occurrences of variable x_i by a_i, that is, $A \downarrow_{X_i} \in c_i$, where $A \downarrow_{X_i}$ denotes the assignment *A* projected to the subset $X_i \subseteq X$ of variables that appear in c_i. It is required that the satisfiability of each constraint be decidable. It is well known that finding a solution to a CSP is an *NP*-complete problem.

2.2. Complete Search Methods for Solving Constraint Satisfaction Problems

There are two main approaches to solving constraint satisfaction problems. The traditional approach is to employ a tree search method to find the solution(s) to a CSP. The basic idea of tree search based approaches is easy to understand. In a nutshell, we first select a variable, and create a choice point on it so that the values in its domain can be tried one after another. For each of these values being tried, we select another variable and create a new choice point on it. This process continues until all variables are exhausted. In this way a *search tree* is formed, with each of its leaves corresponding to either a *solution* or a *failure*. In the case of a failure, the search will continue at the next available node. We call such a process as a *backtracking*. Figure 1 shows a sample search tree for a simple CSP with three variables x_1 , x_2 *and* x_3 associated with domains $d_1 = \{1,2\}$, $d_2 = \{2,3\}$ *and* $d_3 = \{1,2,3\}$, and three constraints $x_1 < x_2$, $x_2 = x_3$ *and* $x_1 + 1 = x_3$. A tick in the leaf indicates that a solution is found, and a cross indicates a failure.

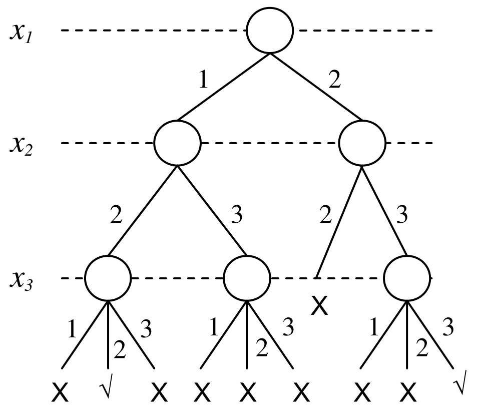


Figure 1. A search tree for the example CSP

Many heuristics 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 to identify and try the most promising values first at each choice point, so that the first (or the next) solution can hopefully be found earlier. Finally, constraint propagation, such as node- and arc-consistency algorithms [14], can be employed at each choice point to reduce the sizes of the variable domains, and thus also the size of the search tree as some of the branches are pruned in the process.

A general framework that incorporates these heuristics is outlined in Figure 2. "⊥" indicates a failure. Note that for clarity purpose we only show the solver that finds the first solution. This framework is the basis

```
Solve(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 $d_{x_i} = \{s_i\}$ ;
  v = var_ordering_heuristics(X);
  repeat {
    a = val_ordering_heuristics( $D'_v$ );
    A=Solve( $\langle X, D', C \cup \{v = a_i\}\rangle$ );
    if ( $A \neq \perp$ ) return A;
    $D'_v = D'_v \setminus \{a\}$ ;
  } until $D'_v = \varnothing$ ;
  return $\perp$ ;
}
```

Figure 2. A general framework of tree search based solvers for CSPs, which finds the first solution

of many commercially available tree search based complete search solvers, such as CHIP [22], ILOG Solver [12] and JSolver [2].

3. Guided Complete Search

Applying value-ordering heuristics is a common and important approach to speed up constraint solving, which can guide the constraint solver to reach the first solution more directionally. However, it is difficult to know what value-ordering heuristics is suitable for different problems, as the effectiveness of value-ordering heuristics is domain-dependent, or sometimes even problem-dependent. The min-conflicts [21] value-ordering heuristics is one of the successful general guidance for complete tree search.

In this section, the Guided Complete Search (GCS) framework for solving CSPs is presented, which coordinates the collaboration of complete tree search (TS) and a collaborating solver in order to produce a complete and efficient CSP solver. In the framework, from the viewpoint of TS, the collaborating solver acts as an "oracle" that generates heuristics for value ordering. Meanwhile, TS narrows the search space for the collaborating solver by constraint propagation after each value commitment. The information exchange operation improves the performance for both solvers in the framework. As a result, a more efficient hybrid solver can be obtained.

A general framework of Guided Complete Search (GCS) is shown in Figure 3. The function the_other_solver_returns() is actually an interface to combine TS and the collaborating CSP solver ("the other solver"), which returns a solution A, a *failure* ("$\perp$"), or an *unknown*. Note that the situation of returning unknown in the collaborating solver usually refers to the case of reaching a pre-set resource limit, such as time limit, maximum number of iterations, *etc*. In such a case the collaborating solver should demonstrate a favor of values of some variables. Hence, the function val_suggestion_from_other_solver() is called, and the variable labeling procedure proceeds in TS. Eventually, solution can be found either in TS or in the collaborating solver.

```
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 $d_{x_i} = \{s_i\}$ ;
  update_the_other_solver( $\langle X, D', C\rangle$ );
  R = the_other_solver_returns();
  if (R=A) return A;
  if (R= $\perp$ ) return $\perp$ ;
  v = var_ordering_heuristics (X);
  repeat {
    a = val_suggestion_from_other_solver();
    update_the_other_solver( $\langle X, D', C \cup \{v = a\}\rangle$ );
    if (the_other_solver_returns()=A) return A;
    if (not the_other_solver_returns()= $\perp$ ) {
      A=GCS( $\langle X, D', C \cup \{v = a\}\rangle$ );
      if ( $A \neq \perp$ ) return A;
      }
    $D'_v = D'_v \setminus \{a\}$ ;
  } until $D'_v = \varnothing$ ;
  return $\perp$ ;
}
```

Figure 3. A framework of GCS solvers for CSPs, which finds the first solution

The cost of employing the collaborating solver to obtain a promising value for a variable might be expensive. A possible way to reduce the cost is shown in Figure 4. We modify the GCS algorithm to provide flexibility for different degree of integration. The function need_suggestion() determines whether there is a need to invoke the other solver, and the function update_the_other_solver() can perform a partial update (from "no-op" to full update) to the other solver in or-

```
GCS*(X, D, C) {
  $\langle X, D', C\rangle$ = constraint_propagation( $\langle X, D, C\rangle$ );
  if ( $\exists d[d \in D \rightarrow d = \emptyset]$ ) return $\perp$ ;
  if ( $\forall d[d \in D \rightarrow | d |= 1]$ )
    return $\{x_i \mapsto s_i \mid i = 1, 2, \ldots, n\}$ , where $d_{x_i} = \{s_i\}$ ;
  v = var_ordering_heuristics(X);
  repeat {
    if (need_suggestion())
      a = val_suggestion_from_other_solver()
    else
      a = val_ordering_heuristics( $D'_v$ );
    update_the_other_solver( $\langle X, D', C \cup \{v = a\}\rangle$ );
    R = the_other_solver_returns();
    if (R=A) return A;
    if (R$\neq\perp$ ) {
      A=GCS*( $\langle X, D', C \cup \{v = a\}\rangle$ );
      if ( $A \neq \perp$ ) return A;
      }
    $D'_v = D'_v \setminus \{a\}$ ;
    } until $D'_v = \emptyset$ ;
  return $\perp$ ;
}
```

Figure 4. A cost-reducing framework of GCS solvers for CSPs, which finds the first solution

der to reduce the communication cost between solvers. Intuitively, the tree search can seek suggestion for value selection whenever a deep backtrack occurs, which implies the default value-order has not been successful and has led to failure.

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_other_solver_returns(). The collaborating solver acts as the value ordering heuristics function to determine which value to instantiate next. Provided the function the_other_solver_returns() always returns within a finite period of time, the GCS framework is sound and complete. This result is shown in the following theorem.

Theorem 1. The GCS framework is sound and complete if the function the_other_solver_returns() always returns in a finite time period.

4. GCS Schemes

In this section, we first present various GCS schemes and their performance figures. We use GCS/*X* to denote the cooperation of TS and an *X* solver, and GCS*/*X* refers to the cost-reducing version of GCS. Note that we use *smallest domain first* variable-ordering heuristics in all experiments.

4.1. GCS/LV

A Las Vegas (LV) method is a randomized algorithm, which can be regarded as an incomplete CSP solver. The idea of Las Vegas solver is simple. It randomly takes sample points in the search space of a problem and validates it. It always returns a correct answer to the problem, but does not guarantee a solution to be found within a time limit. An outline of the Las Vegas solver is shown in Figure 5. If the Las Vegas solver returns *unknown*, the GCS/LV then asks for a value suggestion by calling the function val_suggestion_from_other_solver() for a particular variable, and the LV_Solver returns the current value of the corresponding variable.

```
LV_Solver(X, D, C) {
  while (not all constraints satisfied and not timeout) {
    Generate a random assignment
      $A = \langle x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\rangle$ where $a_i \in d_i$ ;
  }
  if (all constraints satisfied)
    return $\{x_i \mapsto s_i \mid i = 1, 2, \ldots, n\}$ where $d_{x_i} = \{s_i\}$ ;
  else
    return unknown;
}
```

Figure 5. An outline of the Las Vegas Solver

Example: Latin Square

To demonstrate the performance of GCS/LV, we built a prototype implementation with constraint programming library JSolver [2] and JDK version 1.3.1 on a Sun Blade 1000 UNIX workstation. The Latin square problem is used to examine the algorithm. A Latin square problem of order n is to find an $n \times n$ matrix that consists of n sets of the numbers 1 to n arranged in such a way that no orthogonal (row or column) contains the same number more than once. The results shown in Table 1 are the median timing results and number of fails in 20 runs and each run is limited to 15 minutes. A "---" indicates that the problem cannot be solved within the time limit using the undergoing algorithm. The bold figures are the best results among TS, LV and GCS/LV.

```
GA_Solver(X, D, C) {
  $t \leftarrow 0$;
  initialize_new (P($t$), Z, D);
  evaluate (P($t$), C);      // conflicts counting
  while (not Termination-condition) {
    $t \leftarrow t+1$;
    select P($t$) from P($t-1$);
    alter_new (P($t$));
    evaluate (P($t$));
    P($t$) = survive P($t$) and P($t-1$);
    }
 if (all constraints satisfied)
   return $\{x_i \mapsto s_i \mid i = 1,2,\ldots,n\}$ where $d_{x_i} = \{s_i\}$;
 else
   return unknown;
}
```

Figure 6. An outline of the Canonical Genetic Algorithms Solver

Surprisingly, in general, the GCS/LV algorithm outperforms both TS and LV standalone execution in this particular problem both in terms of runtime and number of fails. Besides, it is able to solve all problem instances in this experiment, while the LV cannot.

	Runtime (sec)			Number of fails		
	TS	LV	GCS/LV	TS	LV	GCS/LV
2×2	0.06	**0.01**	0.06	**0**	n/a	**0**
4×4	**0.07**	101.18	**0.07**	**0**	n/a	**0**
6×6	**0.11**	---	**0.11**	**0**	n/a	**0**
8×8	0.23	---	**0.22**	**0**	n/a	**0**
10×10	0.34	---	**0.32**	1	n/a	**0**
12×12	0.75	---	**0.5**	15	n/a	**1**
14×14	0.86	---	**0.58**	20	n/a	**0**
16×16	1.11	---	**0.78**	21	n/a	**0**
18×18	1.44	---	**1.12**	34	n/a	**2**
20×20	7.62	---	**1.27**	153	n/a	**2**
22×22	2.71	---	**1.51**	86	n/a	**1**
24×24	2.59	---	**1.82**	46	n/a	**7**
26×26	8.53	---	**2.13**	314	n/a	**2**
28×28	61.94	---	**2.5**	2003	n/a	**8**
30×30	4.7	---	**2.83**	55	n/a	**2**

Table 1. Comparison among TS, Las Vegas solver and GCS/LV on Latin Square problems

4.2. GCS/GA

The schema theory and building blocks hypothesis [20] of Genetic Algorithms (GA) explain that key *building blocks* are preserved from generation to generation during the genetic search. In GCS/GA, a building block of GA can be interpreted as a branch of subtree in TS. In addition, the TS and GA processes share the same problem context (variables, domains and constraints), therefore when the domain size is reduced by constraint propagation in TS, it also narrows the search space in GA. Basically, The GA solver in GCS/GA is performing a canonical genetic algorithm procedure as shown in Figure 6. Five essential components are specified in GCS/GA, they are: *chromosome structure*, *population initialization*, *fitness evaluation*, *chromosome alteration*, and *termination condition*.

Chromosome Structure, Initialization and Fitness Evaluation

In order to solve CSP with genetic algorithm, the problem has to be encoded in the chromosome form. The value of each variable $x_i \in X$ in the CSP is represented by a gene in the GA, which is an element of d_i. Each chromosome is a sequence of genes corresponding to a valuation for all the variables. If $X = \{x_{1,\ldots,}x_n\}$, then a chromosome $s_1s_2 \cdots s_n$ represents the variable assignment, where $s_i \in d_i$.

A small size of population is initialized, for example less than 20 chromosomes, in order to let the population converge faster. Besides, for those bounded variables, the corresponding genes are pre-set with a value assigned to the variable in TS, and randomly generate a value for the rest genes. Note that the generated value must be in domain.

Solving CPSs can be rewritten as an optimization problem that minimizes the number of constraint violations [16]. Thus, min-conflicts heuristics [21] can be used to guide the generic search in GA_Solver.

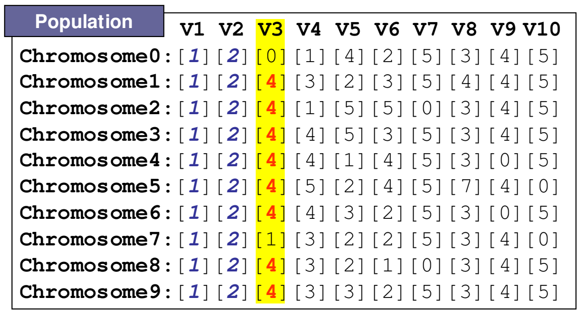
Alteration and Termination

Crossover and mutation are the main genetic operators in GA_Solver. A simple n-point crossover is adopted for the information swapping (local search), n can be determined by the length of chromosome, say n=5 for 50-gene chromosome. The mutation operator is slightly different from the canonical one. It does not mutate the gene that its corresponding variable has been bounded. It is similar to the initialization procedure.

One of the objectives of GA_Solver is to determine which value for a particular variable is better (which sub-tree potentially contains solution). Therefore, the GA_Solver is terminated when a particular gene among the population is converged, that means all genes in the population which represent the same variable are equal to same value. This termination criteria obey the principle of survival-of-fittest, good genes are preserved during evolution. Thus, the converged value is used to suggest GCS/GA for variable instantiation. Note that GA_Solver is also terminated when the maximum number of iteration is reached that ensures the termination of the entire algorithm.

Value Suggestion to GCS with GA

When GCS request a value suggestion for a particular variable, the function val_suggestion_from_other_solver() is called, the GA solver returns a value which is the most popular in the latest population with respect to the corresponding gene. For example, Figure 7 shows a GA population pool with ten chromosomes, the GA solver is being requested for value suggestion for variable V3. In this case, the GA solver returns value "4" as the value suggestion to GCS, since it is the most popular one in the population.



```
Population    V1 V2 V3 V4 V5 V6 V7 V8 V9 V10
Chromosome0:[1][2][0][1][4][2][5][3][4][5]
Chromosome1:[1][2][4][3][2][3][5][4][4][5]
Chromosome2:[1][2][4][1][5][5][0][3][4][5]
Chromosome3:[1][2][4][4][5][3][5][3][4][5]
Chromosome4:[1][2][4][4][1][4][5][3][0][5]
Chromosome5:[1][2][4][5][2][4][5][7][4][0]
Chromosome6:[1][2][4][4][3][2][5][3][0][5]
Chromosome7:[1][2][1][3][2][2][5][3][4][0]
Chromosome8:[1][2][4][3][2][1][0][3][4][5]
Chromosome9:[1][2][4][3][3][2][5][3][4][5]
```

Figure 7. An example of GA population

Examples

We built a prototype implementation of GA and GCS/GA with constraint programming library JSolver [2] and JDK version 1.3.1 on a Sun Blade 1000 UNIX workstation. Two test cases are used to examine the GCS/GA algorithm, which are the *N-Queen problem* and *graph coloring problem*. The following results are median result in 20 runs and each one is limited to 15 minutes.

The N-Queen Problem

The *N-Queen* problem is the problem of placing N queens on an $N \times N$ chessboard so that no queens can take any other. A queen attacks another queen when both of them are placed on the either same row, column or (+ve/-ve) diagonal. We have applied TS, GA and GCS/GA algorithms on a set of *N-Queens* problem instances from N=10 to 150, and the computation performance and the number of fails are shown in Figure 7. The TS algorithm works well up to a hundred queens, but it does not work for the problem thereafter within the time limit. However, the GCS/GA can obtain solution consistently, since least backtracking occurred in the search. And the overhead is due to the computation of fitness in GA_Solver.

	Runtime (sec)			Number of fails		
	TS	GA	GCS/GA	TS	GA	GCS/GA
N=10	**0.04**	0.55	0.18	7	n/a	**3**
N=20	**0.08**	5.2	0.57	29	n/a	**5**
N=30	**0.1**	39.21	1.07	29	n/a	**1**
N=40	**0.11**	264.01	2.44	19	n/a	**2**
N=50	**0.73**	---	4.08	479	n/a	**2**
N=60	**0.73**	---	7.61	400	n/a	**4**
N=70	**0.13**	---	12.94	**1**	n/a	11
N=80	**0.2**	---	16.89	20	n/a	**4**
N=90	**1.05**	---	22.36	335	n/a	**0**
N=100	**0.29**	---	34.5	21	n/a	**5**
N=110	---	---	**46.51**	---	n/a	**6**
N=120	---	---	**59.19**	---	n/a	**2**
N=130	---	---	**81.17**	---	n/a	**21**
N=140	---	---	**89.13**	---	n/a	**1**
N=150	---	---	**113.94**	---	n/a	**2**

Table 2. Comparison among TS, Genetic Algorithms and GCS/GA on *N-Queen* problems

Graph Coloring Problem

The task of solving graph coloring problem is to paint all regions in the graph such that no neighborhood region is sharing the same color. The problem instance we have chosen is appeared in [19]. It is a 110-region graph and the goal is to paint it with four colors; the search space of this problem is *1.68E+66*. Wong *et al.* [19] have designed a specific variable and value ordering heuristics that dedicated for solving this particular problem. In this case, GCS/GA algorithm takes less then 1.1 seconds and visited 36 choice-points to obtain a solution. Although Wong's algorithm takes less then a second to obtain a solution, it has to visit 48 choice-points. When the domain knowledge has been known and transformed as heuristic to guide the constraint solving, it could be very efficient. However, our algorithm have visited less choice-points to reach the solution, which implies that the GCS/GA is more efficient than Wong's, in term of number of backtracking. It can be realized that the guidance provided by GA solver is good enough to direct the search of TS solver to reach a solution.

4.3. GCS/MIP

The GCS/MIP algorithm needs to maintain two models, namely a CSP model and a linear model, during the search process. The linear model is obtained by the transformation methods which will be discussed in

	Runtime (sec)				Number of fails			
	TS	MIP	GCS/MIP	GCS*/MIP	TS	MIP	GCS/MIP	GCS*/MIP
3×3	**0**	0.13	0.02	**0**	**1**	n/a	1	**1**
4×4	**0**	125.85	0.43	**0**	**7**	n/a	97	**7**
5×5	**0.11**	---	15.41	0.2	702	n/a	1382	**53**
6×6	10.47	---	78.06	**2.42**	57246	n/a	3522	**377**
7×7	---	---	648.64	18.92	---	n/a	8292	515
8×8	---	---	496.35	24.69	---	n/a	2606	329
9×9	---	---	---	**109.34**	---	n/a	---	**397**
10×10	---	---	---	**235.26**	---	n/a	---	**376**

Table 3. Comparison among TS, MIP, GCS/MIP and GCS*/MIP on Magic Square problems

the following subsection. It is to map each value in a variable domain into a 0-1 variable. Complete tree search (TS) is applied on the CSP model, while MIP techniques are applied on the linear model. The two search trees are explored in parallel. The solution of the linear relaxation problem is used to guide the value selection of the tree search. At each choice point, from those 0-1 variables corresponding to the current domain of the active variable, we choose the first one with the largest solution value, *i.e.* nearest to one, and use the value corresponding to this variable as the value selection heuristics for the tree search.

It is expensive to solve a linear relaxation problem at every node. A possible improvement is to reduce the number of times that value selection is guided by linear relaxation solutions. Instead of invoking the value selection heuristics at every node, it is invoked only when tree search's default value ordering heuristics proves to fail to find a promising value. Such situation can be identified by a deep backtrack in the tree search.

When using the default variable ordering heuristics, and using deep backtrack as a need-suggestion signal for the cost-reducing framework, we get the GCS/MIP variant, which is denoted by GCS/MIP*.

Linear Formulation of CSP Models

The key point left is how to transform a CSP model into a linear model. The main idea of the transformation method is to map each value in a variable domain into a 0-1 variable. We use $v_{x=a}$ to represent the variable corresponding to a value a in the domain of a variable x. Here, $v_{x=a} = 1$ if $x = a$, and $v_{x=a} = 0$ if $a \notin x$. The domain constraint $x \in D(x)$ is transformed to a linear constraint $\sum_{a\in D(x)} v_{x=a} = 1$, where $v_{x=a} \in \{0,1\}$. In this way, we restrict one and only one of such 0-1 variables, which correspond to the same original finite domain variable, to be equal to 1. In general, we work on incompatible tuples to construct the linear formulation of a constraint. Suppose the constraint involves n variables $x_1,\ldots,x_n$, with an incompatible tuple $(a_1,\ldots,a_n)$, we add one linear constraint $\sum_{i=1}^{n} v_{x_i=a_i} \le n-1$ to the linear model. It achieves the same effect as the original constraint by prohibiting the participating variables from taking corresponding values in an incompatible tuple simultaneously. For some specific kinds of constraints, for example, linear constraint, alldifferent constraint, cardinality constraint, element constraint, etc., better transformation method is available. Please refer to [15] for details and replace those non 0-1 variables by the linking constraint $x = \sum_{a\in D(x)} a \times v_{x=a}$.

Example: Magic Square

A system has been implemented on the top of ILOG CPLEX 8.0 [11] / Solver 5.2 [12] on a Sun Ultra 5/400 UNIX workstation. Experiments are conducted on magic square problems with orders from 3 to 10. The result is listed in Table 3. For TS, we use smallest domain first as the variable ordering heuristics and smallest value first as the value ordering heuristics. For MIP, the model is built from the CSP model by the transformation methods discussed. We use primal simplex algorithm to solve the linear relaxation problems at every node. For GCS/MIP and GCS*/MIP, the variable ordering heuristics is the same as that of TS. For GCS*/MIP, the default value ordering heuristics is the same as that of TS. We use primal simplex algorithm to solve the linear relaxation problems of the hybrid algorithm, too.

From the experiment results, the GCS/MIP algorithms perform better than both TS and MIP approach in general. And GCS*/MIP outperforms the basic GCS/MIP either in terms of fails or in terms of time.

5. Conclusions and Future work

This paper describes a complete and robust hybrid framework for guided complete search for solving general constraint satisfaction problems. To reduce the communication cost between solvers during search, the GCS framework provides various degree of collaboration between solvers. Under the GCS framework, solvers exchange information during search, which enhances the performance of each others. The main contribution of GCS is 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. And the value commitments made by the tree search based solver, as well as other information such as the results of constraint propagation, help the collaborating solver to reduce the problem size in order to speed up the whole search process.

Three different CSP solvers: Las Vegas Solver, Genetic Algorithms and Mixed Integer Programming have been employed by the GCS framework yielding GCS/LV, GCS/GA and GCS/MIP. The experimental evident show that the guidance provided by the other solver is promising and directs the search toward a first solution. Furthermore, the results show that GCS is able to solve certain hard problems without specific prior design or domain knowledge, which outperforms complete tree search and the other solver standalone execution.

For future work, the proposed framework can be extended to solve constraint optimization problem (COP) by implementing branch-and-bound algorithm. Eventually, a generic framework for solving CSP/COP would be obtained.

Acknowledgement

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

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

[2] H.W. Chun, "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*, London, 1999.

[3] H.W. Chun and H.C. Chan, "The Design of a Multi-Tiered Bus Timetabling System", In *Proceedings of the Twelfth International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems*, Cairo, 1999.

[4] H.W. Chun and R.W.T. Mak, "Intelligent Resource Simulation for an Airport Check-in Counter Allocation System", In *IEEE Transactions on Systems, Man, and Cybernetics - Part C: Applications and Reviews*, Vol. 29, No. 3, 1999, pp. 325-335.

[5] S.K.L. Fung and H.W. Chun, "Implementing Virtual Path Assignment Using a Heuristic-Driven CSP Algorithm", In *Proceedings of the 4th Systemics, Informatics and Cybernetics*, Orlando, July 2000.

[6] C.P. Gomes and D. Shmoys, "Completing quasigroup or latin squares: A structured graph coloring problem", In *Prodeeings of Computational Symposium on Graph Coloring and Generalizations*, 2002.

[7] C.P. Gomes and D. Shmoys, "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 (CP-AI-OR'02)*, Le Croisic, France, March 25-27, 2002, pp. 291-305.

[8] J.N. Hooker, H. Kim, and G. Ottosson, "A declarative modeling framework that integrates solution methods", *Annals of Operations Research*, Vol. 104, 2001, pp.141-161.

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

[10] J.N. Hooker, G. Ottosson, E.S. Thorsteinsson, and H. Kim, "On integrating constraint propagation and linear programming for combinatorial optimization", In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, AAAI, The AAAI Press/The MIT Press, July 1999, pp. 136-141.

[11] ILOG Inc., S. A., Gentilly, France. ILOG CPLEX 8.0, User Manual, 2001.

[12] ILOG Inc., S. A., Gentilly, France. ILOG Solver 5.2, User Manual, 2001.

[13] J.H.M. Lee, H.F. Leung, P.J. Stuckey, V.W.L. Tam, and H.W. Won, "Using Stochastic Methods to Guide Search in CLP: a Preliminary Report", *1996 Asian Computing Science Conference*, Springer-Verlag, LNCS 1179, Singapore, 1996, pp. 43-52.

[14] A. Mackworth, "Consistency in networks of relations", *Artificial Intelligence*, 1977, 8(1), pp. 99-118.

[15] P. Refalo, "Linear formulation of constraint programming models and hybrid solvers", In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP-00), volume 1894 of Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2000, pp. 369-383.

[16] M.C. Riff, "Evolutionary Algorithms for Constraint Satisfaction Problems", In *Proceedings of XVIII International Conference of the Chilean Computer Science Society SCCC'98*, 1998.

[17] V. Tam and P. Stuckey, "An efficient heuristic-based evolutionary algorithm for solving constraint satisfaction problems", In *Proceedings of 3rd IEEE Symposium on Intelligence in Neural and Biological Systems (INBS)*, Washington DC, May, 1998.

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

[19] G.Y.C. Wong and H.W. Chun, "CP Heuristics: MWO/FFP Hybrid and Relaxed FFP", In *Proceedings of the 4th Systemics, Informatics and Cybernetics*, Orlando, July 2000.

[20] D.E. Goldberg, "Genetic Algorithm in Search, Optimization and Machine Learning," Addison-Wesley Pub. Co., Inc., 1989.

[21] S. Minton, M.D. Johnston, A.B. Philips and P. Laird, "Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems", In *Artificial Intelligence*, Vol. 58, 1992, pp. 161-205.

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