ZDC-Rostering: A Personnel Scheduling System Based On Constraint Programming

Edward Tsang, John Ford, Patrick Mills, Richard Bradwell, Richard Williams, Paul Scott

Technical Report 406
4 June 2004
Department of Computer Science
University of Essex
Colchester CO4 3SQ
United Kingdom
http://cswww.essex.ac.uk/CSP/

Abstract

Personnel scheduling is a very practical problem. It is widely studied because solutions to it can be generalized to many other problems. This paper describes ZDC-Rostering, a powerful constraint-based tool for personnel scheduling. ZDC-Rostering is based on a computer-aided constraint programming package called ZDC, which decouples problem formulation (or modelling) from solution in constraint satisfaction problems, and provides a set of constraint satisfaction algorithms, including complete and incomplete algorithms, to users who are not required to learn how these algorithms work. ZDC allows us to focus on constraint modelling in the rostering problem, which is expressed by a simple declarative language called EaCL. The simplicity of EaCL allows users with basic training in programming and problem specification to add new constraints easily. Solvers supplied in ZDC include a generalized Forward Checking solver, a Linear Programming solver and local search solvers implementing Guided Local Search, Tabu Search and Genetic Algorithms. Our experiments show that Guided Local Search is capable of solving realistic and very tightly constrained problems efficiently.

1. Introduction

Personnel scheduling is a problem that is encountered in many situations, such as hospitals, airline crews and factories. It has been widely studied because techniques applicable to these problems can be generalized to other combinatorial problems.

So-called "nurse rostering" problems (which require the assignation of shifts to personnel holding the correct professional qualifications, subject to satisfying regulatory and other types of hard and soft constraints) are a particular category of rostering problem that has been studied for a number of years. Such problems present difficulties for the constraints practitioner in two distinct ways: first, in devising an effective model of the problem and, second, in its efficient solution. In accordance with the main thrust of the present paper, we will focus substantial attention on the first of these issues before proceeding to consider the second. In the meantime, however, we note some of the recent approaches to efficient solution of such problems: genetic algorithms [Aickelin and Dowsland, to appear], variable neighbourhood search [Burke et al 2003], tabu search [Burke et al 1999], fuzzy constraints [Meyer auf'm Hofe 2001a and 2001b], and hybrid methods [Li et al 2003; Burke et al 2001].

Many such problems can be formulated as Constraint Satisfaction Problems (CSP) or Constrained Optimization Problems (COP) [Tsang 1993; Freuder & Mackworth 1994]. Constraint satisfaction is a powerful technique which has been successfully applied to scheduling problems, e.g. see [Lever et al 1995; Rodosek & Wallace 1998; Hnich et al 2002; Bourdais et al 2003]. In general, these techniques use constraints in the problem to prune the search space (especially in complete search) or guide the search (especially in stochastic search).

Within the Constraint Satisfaction/Optimization research community a large amount of effort has been invested in engineering stronger algorithms, studying problem difficulty and more recently studying the implications of using different problem formulations. As a result, individual researchers, and the research community as a whole, have accumulated a large amount of implicit and explicit domain knowledge regarding how best to solve problems using constraint technology. Nevertheless, it is difficult to transfer the technology to an industrial setting without requiring an expert in the field, when one bears in mind the knowledge required to apply constraint technology effectively.

The knowledge required to apply constraint technology effectively includes:

- Knowing how to formulate a given problem as a CSP/COP
- Knowing how to incorporate domain knowledge into the solver
- Knowing how to choose a good formulation for a given problem
- Knowing which solver to apply to a given problem
- Knowing how to engineer a solver.
-

Various industrial strength packages, such as ILOG Solver, have been implemented with the explicit aim of making access to constraint technology easier (http://www.ilog.fr) [Michel & Van Hentenryck 2001]. Even these packages require a significant amount of expertise to use because they usually come in the form of a constraint library that can be linked to a standard application, written in the desired 3GL. Knowledge of the target language and the constraint library is still required. Generally they concentrate only on the issue of solving, relieving the user of the burden of writing their own solver.

The CACP project attempts to provide a system that encompasses the entire process of applying constraint technology. It supports the tasks of problem formulation and entry, in addition to supplying pre-written solvers and aiding the user in choosing which of the available algorithms to apply. Problems are modelled in the declarative language EaCL (standing for Easy abstract Constraint Language) [Mills et al 1998], which the user can enter via an intuitive user interface. The problem specification is decoupled from the solvers, so that users may experiment with different problem formulations easily. Careful attention has been directed to providing a user-friendly GUI-based system for easy entry of problem constraints. An extensive help system is also provided. Having formulated the problem in the EaCL language and entered it, the user can solve the problem by using one of the pre-written generic solvers. Choosing the correct solver from a problem is often a difficult task and therefore the CACP project makes an initial attempt to address this issue also.

Within the research community, competition between algorithms drives researchers to produce ever-faster algorithms that produce better results, for a restricted set of problems. Generally this is accomplished by tailoring the algorithm using large amounts of domain knowledge. The goal of the CACP project was not to compete with these highly specialized algorithms. Rather, the goal was to provide a simple, easy-to-use system to enable researchers and users to produce solutions without having to invest much effort to learn the constraint language, constraint solving techniques or how to use the system. The system implemented is primarily targeted for users who are not interested in implementing constraint programming techniques, but would nevertheless like to exploit constraint technology for their own benefit.

2. The ZDC Architecture

ZDC-Rostering is built upon the ZDC constraint modelling system. The ZDC architecture is illustrated in Figure 1. The main flow of control starts with the entry of the problem definition in the EaCL language, developed within the group. Additional data, required for more demanding problems, can be read automatically from Excel by using it as an automation server. Problem description entry is performed using either the ZDC or ZDC Direct user interface. ZDC Direct allows the user to enter the problem formulation as text. ZDC uses a more elaborate interface to shield the user from the EaCL grammar, making problem entry easier. In ZDC-Rostering, the application is written in Prolog. It generates (from the user's problem specification) a constraint model in EaCL syntax, which is then input to ZDC Direct for solving.

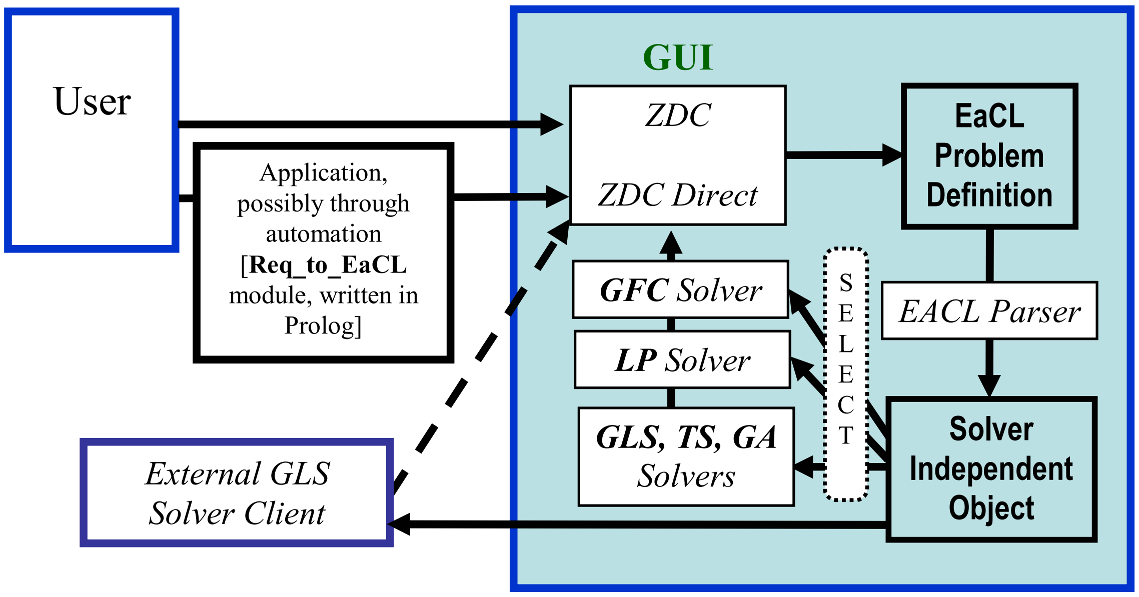
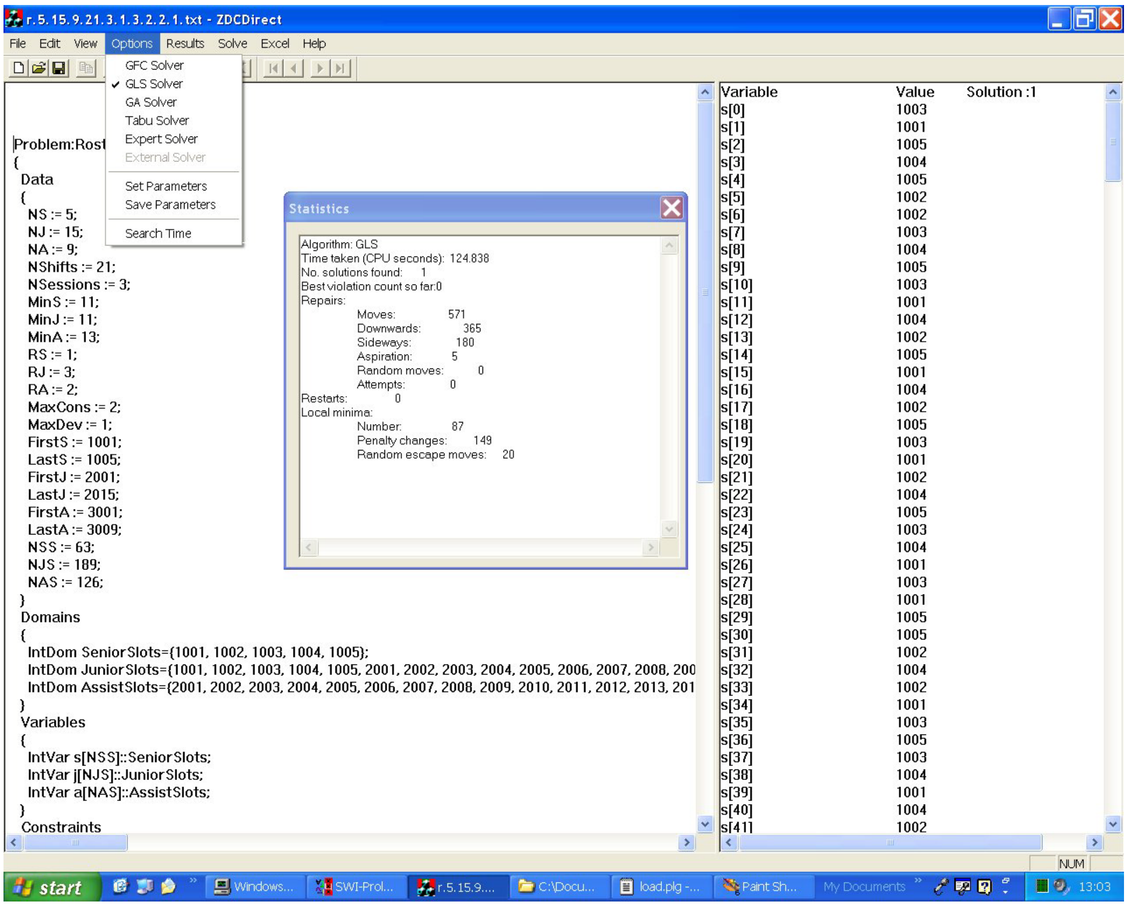

User
Application, possibly through automation [Req_to_EaCL module, written in Prolog]
GUI
ZDC
ZDC Direct
EaCL Problem Definition
GFC Solver
LP Solver
GLS, TS, GA Solvers
SELECT
EACL Parser
Solver Independent Object
External GLS Solver Client

Figure 1: The ZDC Architecture, the foundation of ZDC-Rostering



```
Problem:Rost
{
 Data
 {
  NS := 5;
  NJ := 15;
  NA := 9;
  NShifts := 21;
  NSessions := 3;
  MinS := 11;
  MinJ := 11;
  MinA := 13;
  RS := 1;
  RJ := 3;
  RA := 2;
  MaxCons := 2;
  MaxDev := 1;
  FirstS := 1001;
  LastS := 1005;
  FirstJ := 2001;
  LastJ := 2015;
  FirstA := 3001;
  LastA := 3009;
  NSS := 63;
  NJS := 189;
  NAS := 126;
 }
 Domains
 {
  IntDom SeniorSlots={1001, 1002, 1003, 1004, 1005};
  IntDom JuniorSlots={1001, 1002, 1003, 1004, 1005, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 200
  IntDom AssistSlots={2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 201
 }
 Variables
 {
  IntVar s[NSS]::SeniorSlots;
  IntVar j[NJS]::JuniorSlots;
  IntVar a[NAS]::AssistSlots;
 }
 Constraints
```

```
Algorithm: GLS
Time taken (CPU seconds):  124.838
No. solutions found:     1
Best violation count so far:0
Repairs:
            Moves:              571
            Downwards:              365
            Sideways:            180
            Aspiration:           5
            Random moves:           0
            Attempts:           0
Restarts:           0
Local minima:
            Number:            87
            Penalty changes:         149
            Random escape moves:     20
```

```
Variable                Value     Solution :1
s[0]                    1003
s[1]                    1001
s[2]                    1005
s[3]                    1004
s[4]                    1005
s[5]                    1002
s[6]                    1002
s[7]                    1003
s[8]                    1004
s[9]                    1005
s[10]                   1003
s[11]                   1001
s[12]                   1004
s[13]                   1002
s[14]                   1005
s[15]                   1001
s[16]                   1004
s[17]                   1002
s[18]                   1005
s[19]                   1003
s[20]                   1001
s[21]                   1002
s[22]                   1004
s[23]                   1005
s[24]                   1003
s[25]                   1004
s[26]                   1001
s[27]                   1003
s[28]                   1001
s[29]                   1005
s[30]                   1005
s[31]                   1002
s[32]                   1004
s[33]                   1002
s[34]                   1001
s[35]                   1003
s[36]                   1005
s[37]                   1003
s[38]                   1004
s[39]                   1001
s[40]                   1004
s[41]                   1002
```

Figure 2: A snapshot of ZDC Direct 1.8, showing the graphics interface with a problem loaded (left), a solution (right) and the statistics window (middle)

Figure 2 shows a snapshot of ZDC Direct 1.8 "in action". A problem, written in EaCL (such as the one shown in Appendix B, which was generated from the Req_to_EaCL module of ZDC-Rostering), can be loaded into the system. A click of the red "1" button (hidden by the Options menu) would find the first solution using the default solver (which can be user-defined), which is then shown in the right-hand frame. A statistics window shows the relevant statistics related to the algorithm selected. The user may modify the problem (e.g. by adding or relaxing constraints) and re-solve, if so desired. He/she can also choose the solver to use (under the Options menu – the simplex solver is not shown in the menu as it will be invoked automatically if the problem is detected to be a linear programming problem). Standard Windows conventions, such as on-line "Help" are adopted. Output can be displayed in Excel grids for easy export.

Once a problem description has been entered by the user, the EaCL parser parses the description. An invalid formulation is reported back to the user via the user interface. A correctly parsed definition generates the raw, solver-independent constraint objects. These solver-independent objects are used by the selected solver as a guide to generate its own set of constraint objects. This architecture essentially separates the solver and its object library from the parser, allowing the architecture to be easily extensible. It is important to allow this separation because EaCL potentially supports a large number of solvers for different problem types. Some of these solvers specialise in solving a particular class of problem and therefore require a different set of constraint objects from solvers dealing with other problem types. Solvers supplied in ZDC include a generalized Forward Checking solver, a Linear Programming (LP) solver and local search solvers implementing Guided Local Search (GLS), Tabu Search (TS) and Genetic Algorithms (GA).

An algorithm selection expert module is responsible for matching the problem formulation to the available solver algorithms that can potentially be applied to that formulation. A solver, when invoked, runs in a separate thread, with only one thread executing at a time. The thread terminates once the solver has found a solution or the solver has been terminated prematurely. The results from the solver are passed back to the user interface, where they are relayed to the user. What we have described above is the normal usage of the system. ZDC can also perform the role of an automation server. This means that stand-alone applications with specific user interfaces can be written for a particular domain, yet as automation clients they can access some of the problem-solving capabilities of ZDC.

ZDC can also perform the role of a problem description server, using sockets. This means that ZDC can provide an external solver with a description of the variables, domains and constraints of the currently parsed problem. This facilitates a greater separation between the two phases of problem modelling and solving. The EaCL Parser generates a solver-independent description of the problem. The description instructs the external client solver how to generate a constraint object tree, using its own constraint object library.

The separation of modelling and solving in ZDC enables problem solvers to focus on modelling the problem (without worrying about how to solve it) and constraint algorithm designers to develop solvers (without having to worry about building user interfaces). In ZDC, it is very easy to add additional solvers to the system. To add a new solver (which may implement a new search algorithm) to ZDC, all the algorithm developer has to do is to build an interface to input solver-independent objects. External solver clients may register with the server so that the server becomes aware of their existence. An external solver can submit itself as a slave solver by sending an appropriate message to the server. As a slave, the external solver is under the direct control of the server. The user can interact with the ZDC interface and force an external solver to solve the current problem being modelled and then return the results, just as if it was an internal solver.

3. Problem Modelling in ZDC-Rostering

To solve a rostering problem, ZDC-Rostering takes the following steps:

Step 1. Problem specification – this comes from domain knowledge;
Step 2. Constraint modelling, or problem formulation – this involves defining the variables, domains and constraints of the problem based on the problem specification;
Step 3. Expressing the problem in EaCL – this requires knowledge of the constraint language;
Step 4. Solving the problem – this is done by calling ZDC's library solvers.

These will be described in the following subsections.

3.1 Problem Specification

Specification of the problem comes directly from domain knowledge. In ZDC-Rostering 1.0, a problem is defined by the following facts:

- The number of senior staff, junior staff and assistants are fixed (NS, NJ and NA);
- The number of shifts (NShifts); for example, given that each week has 7 days, each of which has 3 shifts, one has to schedule 21 shifts in a week;
- The number of parallel sessions (NSessions) per shift; for example the team may be required to serve 3 wards;
- The number of senior staff, junior staff and assistants required for each session (RS, RJ and RA); senior staff may be used to fill junior slots, but not vice versa; junior staff may be used to fill assistant slots, but not vice versa;
- No one is allowed to work for more than a specified number (e.g. 2) of consecutive sessions (MaxCons);
- No one must work for fewer than a specified minimum number of sessions; this minimum is calculated as the average load minus a specified number MaxDev (e.g. 1).

An example of a problem specification is shown in Appendix A. The problem defined here is a generic one. More realistic constraints, such as the availability of individual staff, compatibility between leaders, varying number of sessions per shift, etc., can be specified as required.

3.2 Constraint modelling, or problem formulation

Modelling is recognised as a frontier of constraint research [Freuder 99]. This is the step which required most of ZDC-Rostering's development time.

To define a constraint satisfaction problem in ZDC, one needs to define the variables (Z), the domains (D) and the constraints (C) (hence the name of the software). To model the rostering problem defined above, three arrays of variables are defined:

- s[0 .. NShifts*NSessions*RS – 1] for senior slots
- j[0 .. NShifts*NSessions*RJ – 1] for junior slots
- a[0 .. NShifts*NSessions*RA – 1] for assistant slots.
- The domains are defined as follows:
- The domain of s[i] for all i is the set of senior staff;
- The domain of j[i] for all i is the union set of senior and junior staff (because senior staff are allowed to serve junior sessions);
- The domain of a[i] for all i is the union set of junior staff and assistants (because junior staff are allowed to serve assistant sessions).

To formulate the constraint that no-one can work on two jobs at the same time, the AllDifferent constraint has been used. It could equally be expressed as follows:

```
Forall t in [0 .. NShifts - 1]
{
  Forall ( i in [t*NSessions*RS .. (t+1)* NSessions*RS-1],
        j in [t* NSessions*RS .. (t+1)* NSessions*RS-1], i < j )
  {
           s[i] <> s[j];
  }
   … <similar constraints for senior, junior and assistant slots> …
}
```

To formulate the constraint that no one can work for more than k consecutive sessions:

```
Forall t in [0 .. NShifts-k-1]
{
  Forall ( sf in [1 .. NS] )
   {
     Count( [ s[t* NSessions*RS], ..., s[(t+k+1)* NSessions*RS-1],
         j[t*NSessions*RJ], ..., j[(t+k+1)*NSessions*RJ-1]], [sf]) <= k;
   }
         … <similar constraints for junior and assistant slots> …
}
```

An appropriate index was needed to enumerate all the posts of all the sessions at the same time. Here t counts the time from 0 to NShifts-1, the number of shifts in the problem. Then it is a matter of counting the number of posts that take the same staff, and making sure that the sum does not exceed k. The above example deals with senior staff only. Here they are assumed to be numbered between 1 and NS, the number of senior staff available.

To formulate the constraint that no senior staff should work for k sessions fewer than a specified workload:

```
Forall ( sf in [1 .. NS] )
{
     Count( s, [sf] ) + Count( j, [sf] ) >= MinS;
}
```

Similar constraints for junior staff and assistants can be defined similarly.

3.3 Coding in EaCL

The Easy abstract Constraint Programming Language (EaCL) is the language used to formulate problems prior to solving. The formulation above is close to, but not exactly in, the syntax of EaCL. A valid problem formulation in EaCL is split into data, domains, variables, constraints and optimization sub-sections. EaCL supports variables with integer, Boolean, real and sets as domains. The EaCL grammar supports a wide range of logical, integer, set and symbolic constraints. There are also various facilities supporting lists and sets as well as conditional branching. A complete description of the EaCL language is available on-line [Mills et al 1998]. Details of the rostering problem, as expressed in EaCL, will not be shown here. Interested readers may find an example in Appendix B.

The Req_to_EaCL module in ZDC-Rostering is responsible for translating the requirement, based on the problem specification, to EaCL. Once the specification is complete, implementing the Req_to_EaCL module is relatively straightforward. ZDC was designed for usability. The (graphical) on-line support in ZDC helps users to learn the EaCL language. The simplicity and expressive power of EaCL made Req_to_EaCL relatively easy to implement. For example, the quantification and nested quantification supported by EaCL helped to reduce the length of the problem file. Req_to_EaCL was written in roughly 200 lines of Prolog code (excluding comments).

3.4 Solving the problem

Once the problem has been written in EaCL, it can be loaded into ZDC-Direct. ZDC provides the users with a number of solvers.

First, a simplex algorithm is used to solve linear problems containing variables with real domains. A Generalized Forward Checking (GFC) algorithm has also been implemented, with a corresponding library of constraint objects, to represent the category of complete search algorithms. The forward checking algorithm, when applied to an optimization problem, maintains the best solution cost and uses it as a bound on the current solution cost. GFC has also been extended so that it can be applied to problems involving n-ary constraints, where n is greater than 2. Built into the GFC algorithm is a thrashing-detection mechanism that detects if the solver is an inefficient method for solving the current problem [Borrett et al 1996]. Upon detection of thrashing the GFC is automatically terminated.

Three local search techniques have been implemented, all sharing the same library of constraint objects. The solvers implemented are a Genetic Algorithm, Tabu Search (TS) and Guided Local Search (GLS). GLS was found to be the most efficient solver for the rostering problem that we have generated so far. The TS implemented shares most of its modules with GLS. It should be emphasized a simple TS, incorporating a taboo list with user-definable length, has been implemented in ZDC.

Guided Local Search (GLS) is a meta-heuristic, first invented by Voudouris [Voudouris 1997; Voudouris & Tsang 1999], and then extended by Mills [Mills 2002; Mills et al 2003]. When the hill climber is caught in a local minimum, GLS provides a means of escaping the local minimum. Essentially, it escapes from a local minimum by adding extra penalty terms to the cost function. When the algorithm detects that it is in a local minimum, it chooses a feature of the current solution to penalize. A term is then added to the cost function to increase the cost of any solution containing the penalized feature. Penalizing the feature results in an increase in the cost of the local minimum. Neighbouring solutions that do not exhibit the penalized features become more desirable, and hill climbing re-commences. The version of GLS implemented in ZDC 1.8 is the Extended GLS (EGLS) developed by Mills, which incorporates aspiration moves and random moves [Mills 2002]. GLS searches on the augmented cost function, which is equal to the original cost function plus the penalty terms. An aspiration move is a move to a new solution which has a lower cost, in terms of the original cost function, than the best solution found so far. ZDC 1.8 implements two types of random moves in GLS. The first is a random move with a certain (normally low) probability. It was found to be useful in problems where the weight of the penalty is set too low. The second type of random move is invoked only at a local minimum: instead of using the standard moves after penalties are applied, random moves are used, with a (normally) low probability, to escape the local optimum.

Users without knowledge of constraint programming may choose to rely on the "Expert Solver" provided in the CACP project, which will initially assign a random solver to the given problem, but learns (over time) the success rate of different solvers to the user. Based on the heuristic that the user is likely to solve similar problems over time, the Expert Solver selects solvers probabilistically according to their past success rate.

4. Problem-solving in ZDC-Rostering

4.1 Problems tackled and the size of their search space

A problem is characterized by a vector:

((NS, NJ, NA), NShifts, NSessions, (RS, RJ, RA), MaxCons, MaxDev)

where NS, NJ and NA are, respectively, the number of senior staff, junior staff and assistants available. NShifts is the number of shifts to schedule and NSessions the number of sessions per shift. RS, RJ and RA senior staff, junior staff and assistants, respectively, are required per session. No one should work for more than MaxCons consecutive sessions. No one should work fewer than the average workload (in terms of sessions) minus MaxDev sessions.

In the focussed set of experiments, we asked ZDC-Rostering to produce weekly schedules, where there are three shifts per day (so NShifts = 7 x 3 = 21). We assumed that there are three sessions per shift (Nsessions = 3); each session requires one senior staff, three junior staff and two assistants. The number of staff available is adjusted to vary the tightness of the problem. In other words, we have tackled a family of problems with the following settings:

((NS, NJ, NA), 21, 3, (1, 3, 2), 2, 1).

The size of the search space in a constraint satisfaction problem is $\prod_{x} |D_x|$, where D_x is the domain of variable x and $| D_x |$ is its size. For example, when there are 5 senior staff, 16 junior staff and 12 assistants in the above problem, the size of the search space is calculated as follows:

- Number of senior slots to fill: NSS = 21 (shifts) x 3 (sessions per shift) x 1 (staff per session) = 63
- Number of junior slots to fill: NJS = 21 (shifts) x 3 (sessions per shift) x 3 (staff per session) = 189
- Number of assistant slots to fill: NAS = 21 (shifts) x 3 (sessions per shift) x 2 (staff per session) = 126
- Domain size for each senior slot i: $D_{s[i]} = 5$
- Domain size for each junior slot i: $D_{j[i]} = 5 + 16 = 21$ (as senior staff can serve junior slots)
- Domain size for each senior slot i: $D_{a[i]} = 16 + 12 = 28$ (as junior staff can server assistant slots)

Therefore, the size of the search space is 5^{63} x 21^{189} x 28^{126}, which is roughly 10^{476}, and thus yields a problem of considerable size.

4.2 Solver Selection

Experiments were conducted on a PC with an Athlon 2500+, 1GB RAM, running under Windows XP.

When the problem ((5, 16, 12), 21, 3, (1, 3, 2), 2, 1) was fed to the Generalized Forward Checking Solver (which conducts a complete search), no solutions were found in over 16 hours. Neither could the Tabu Search Solver and the Genetic Algorithm Solver manage to find solutions in 16 hours. Only the Guided Local Search (GLS) Solver was able to solve this problem, which it did within 1 CPU minute. Therefore, in the rest of the experiments reported here, GLS was used for all problems.

It is important to emphasize again here that the versions of Tabu Search and Genetic Algorithm implemented were fairly basic. For example, aspiration has not been made available in the Tabu Search. Therefore, the results reported here should not be regarded as a beauty-contest between the above-mentioned stochastic search algorithms.

4.3 Experimental Results

The size of a constraint satisfaction problem is only one of the factors that affect the difficulty of the problem, but it is not the most significant factor. The tightness of the problem also determines the difficulty of a problem [Cheeseman et al 1991]. We have tested ZDC-Rostering with problems of increasing tightness. We focus on problems of the class ((5, 16, A), 21, 3, (1, 3, 2), 2, 1), with decreasing A, starting with A=12.

What we found was consistent with the experience of other constraint researchers. For loose problems, the time taken by GLS increased gradually, but was of the same order of magnitude. GLS is stochastic in nature. Problems with the number of assistants (A) between 12 and 9 roughly took one CPU minute, with a standard deviation of below 7 seconds. However, when A is reduced to 8, the time taken by ZDC-Rostering ranged from 195 CPU seconds to failing after one CPU hour. When A is reduced to 7, no solution was found after one CPU hour. So it is reasonable to assume that ((5, 16, 8), 21, 3, (1, 3, 2), 2, 1) is close to phase transition [Hogg & Williams 1994; Smith & Grant 1995]. As reported in the literature, for problems of similar size, computation cost increases sharply as one approaches phase transition. Figure 3 shows the minimum time taken by ZDC-Rostering in solving the problems discussed.

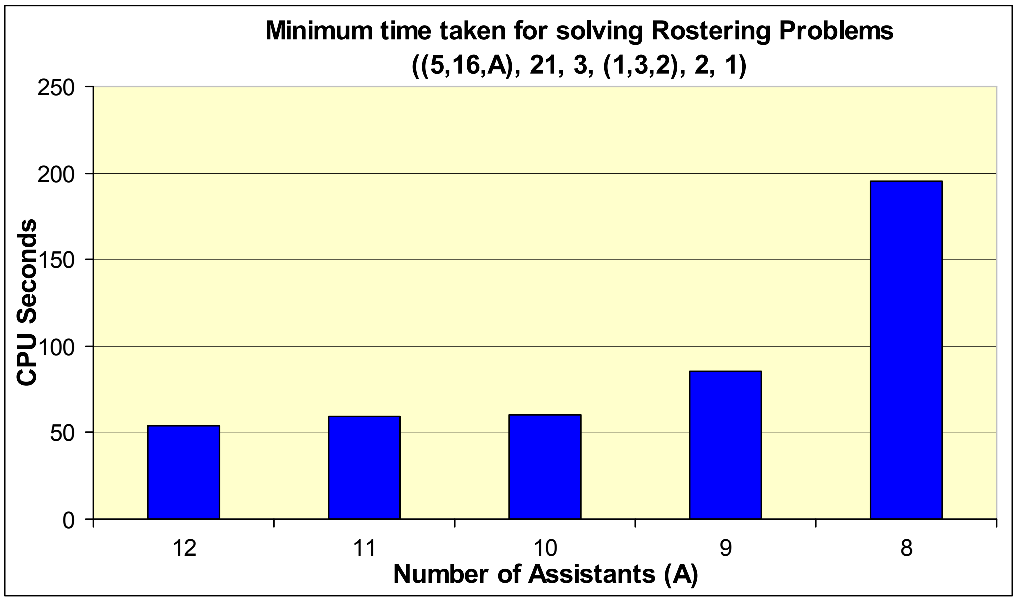


Figure 3: Minimum time taken for solving rostering problems of increasing tightness

Next, we investigate the impact of increasing the problem size. When we increase the number of shifts in the problem, the number of variables grows. Consequently, the size of the neighbourhood grows. The major impact on GLS is that it takes more time to explore the neighbourhood. Like other local search algorithms, GLS examines $\sum_x |D_x|$ neighbours in each move. Therefore, as the number of variables grows, we expect GLS to take more time per move, with a roughly linear increase.

In one of the runs in solving ((5, 16, 12), 21, 3, (1, 3, 2), 2, 1), GLS took 62 CPU seconds and 243 moves to solve the problem, which means it took 0.25 seconds per move. When the number of shifts is doubled, GLS took 264 CPU seconds and 512 moves to solve the problem, which means it took 0.52 seconds per move. When the number shifts is doubled, GLS took 604 CPU seconds and 799 moves to solve the problem, which means it took 0.76 seconds per move. As expected, the number of moves required to find solutions increases as the number of shifts increase. The time required per move increases roughly linearly with the number of shifts, which is consistent with our expectation.

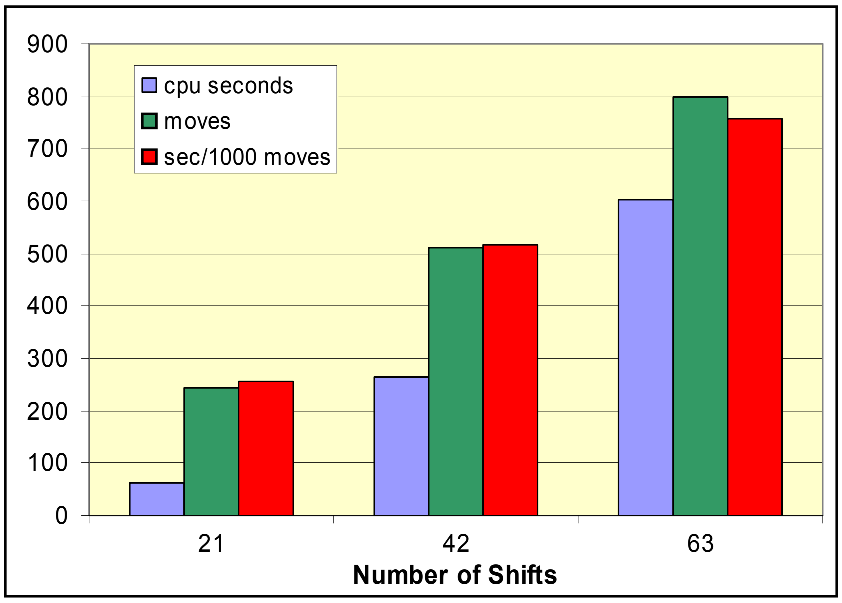


Figure 4: Increase in search time per move in GLS as the number of shifts increases

5. Conclusions

In this paper, we have presented a new rostering system, ZDC-Rostering. It comprises a Req_to_EaCL module to be used in conjunction with ZDC Direct. Req_to_EaCL translates rostering requirements into constraint satisfaction problems, expressed in the EaCL grammar. After loading the problem into ZDC Direct, the user may attempt to solve the problem using the solvers provided. We have demonstrated in this paper that the ZDC and ZDC Direct architecture allows system developers to focus on problem specification (formally defining the requirements), problem formulation (defining the variables, domains and constraints) and problem solving (finding solutions for CSPs) independently.

Equipped with Extended Guided Local Search (GLS), ZDC-Rostering is capable of solving (possibly tightly constrained) problems of realistic size within reasonable time. Thanks to the modularity of ZDC-Direct, the constraints considered so far have been implemented reasonably easily in the Req_to_EaCL module. The EaCL grammar is designed to be simple and easy to learn. New constraints, such as the availability of individual staff, can be added easily, either through Req_to_EaCL, or directly in ZDC-Direct (which would be useful for rescheduling). To summarize, building on a sound and powerful system, ZDC-Rostering has full potential to solve realistic rostering problems.

Acknowledgements

We wish to thank both Nathan Barnes and James Borrett for their contribution to the CACP project. This project was funded by the EPSRC grant GR/L20122 and a University of Essex Research Promotion Fund grant

References

Aickelin, U. & Dowsland, K.A., An indirect algorithm for a nurse scheduling problem, Computers and Operational Research, to appear.

Borrett, J.E., Tsang, E.P.K. & Walsh, N.R., Adaptive constraint satisfaction: the quickest first principle, Proceedings, 12th European Conference on AI, Budapest, Hungary, 1996, 160-164

Bourdais, S., Galinier, P. & Pesant, G., HIBISCUS: a constraint programming application to staff scheduling in health care, in Rossi, F. (ed.), Proceedings, 9th Principles and Practice of Constraint Programming (CP 2003), 2003, 153-167

Bradwell, R., Ford, J., Mills, P., Tsang, E.P.K. & Williams, R., An overview of the CACP project: modelling and solving constraint satisfaction/optimisation problems with minimal expert intervention, Workshop on Analysis and Visualization of Constraint Programs and Solvers, Constraint Programming, Singapore, 2000

Burke, E.K., Causmaecker, P. De, & Vanden Berghe, G., A hybrid tabu search algorithm for the nurse rostering problem, B. McKay et al (Eds.): Simulated Evolution and Learning 1998, LNCS 1585, Springer Verlag, 1999, 187-194

Burke, E.K., Causmaecker, P. De, Cowling, P., & Vanden Berghe, G., A memetic approach to the nurse rostering problem, Applied Intelligence special issue on Simulated Evolution and Learning, Springer Verlag, Vol 15, 2001, 199-214

Burke, E.K., Causmaecker, P. De, Petrovic, S., & Vanden Berghe, G., Variable neighbourhood search for nurse rostering problems, M.C.G. Resende & J. Pinho de Sousa (Eds.), Metaheuristics: Computer Decision-making, Chapter 7, Kluwer, 2003, 153-172

Cheeseman, P., Kanefsky, B. & Taylor, W.M., Where the really hard problems are, Proc., 12th International Joint Conference on AI, 1991, 331-337

Freuder, E.C. & Mackworth, A., (ed.), Constraint-based reasoning, MIT Press, 1994

Freuder, E.C., Modeling: the final frontier, The First International Conference on The Practical Application of Constraint Technologies and Logic Programming (PACLP), London, April 1999, 15-21

Glover, F., Tabu search Part I, Operations Research Society of America (ORSA) Journal on Computing 1, 1989, 109-206

Glover, F., TABU search and adaptive memory programming -- advances, applications and challenges, in Barr, Helgason & Kennington, (ed.), Interfaces in Computer Science and Operations Research, Kluwer Academic Publishers, 1996

Goldberg, D.E., Genetic algorithms in search, optimization, and machine learning, Reading, MA, Addison-Wesley Pub. Co., Inc., 1989

Haralick, R.M. & Elliott, G.L., Increasing tree search efficiency for constraint satisfaction problems, Artificial Intelligence, Vol.14, 1980, 263-313

Hnich, B., Kiziltan, Z. & Walsh, T., Modelling a balanced academic curriculum problem, Proceedings, Fourth International Workshop on Integration of AI and OR Techniques in Constraint programming for combinatorial optimisation Problems, CP-AI-OR'02, 2002, 121-131

Holland, J.H., Adaptation in natural and artificial systems, University of Michigan press, Ann Arbor, MI, 1975

Hogg, T. & Williams, C.P., The hardest constraint problems: a double phase transition, Research Note, Artificial Intelligence, Vol.69, 1994, 359-377

Lever, J., Wallace, M. & Richards, B., Constraint logic programming for scheduling and planning, British Telecom Technology Journal, Vol.13, No.1., Martlesham Heath, Ipswich, UK, 1995, 73-80

Li, H., Lim, A. & Rodrigues, B., A hybrid AI approach for nurse rostering problem, Proceedings of the 2003 ACM Symposium on Applied Computing, ACM Press, 2003, 730-735

Meyer auf'm Hofe, H., Solving rostering tasks as constraint optimization, E. Burke & W. Erben, Practice and Theory of Automated Timetabling 2000, LNCS 2079, Springer Verlag, 2001, 191-212

Meyer auf'm Hofe, H., Nurse rostering as constraint satisfaction with fuzzy constraints and inferred control strategies, E.C. Freuder and R.J. Wallace (Eds.), Constraint Programming and Large Scale Optimisation Problems, DIMACS Series, Vol 57, AMS, 2001, 67-99.

Michel, L. & Van Hentenryck, P., OPL++: a modeling layer for constraint programming libraries, Proceedings, Third International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR-01), Wye, UK, 8-10 April 2001, 205-219

Mills, P., Tsang, E.P.K., Williams, R., Ford, J. & Borrett, J., EaCL 1.0: an easy abstract constraint programming language, Technical Report CSM-321, University of Essex, Colchester, UK, December, 1998

Mills, P., Extensions to guided local search, PhD Thesis, Department of Computer Science, University of Essex, Colchester, UK, 2002

Mills, P., Tsang, E.P.K. & J.Ford, J., Applying an extended guided local search to the quadratic assignment problem, Annals of Operations Research, Kluwer Academic Publishers, Vol.118, 2003, 121-135

Rodosek, R. & Wallace, M., A generic model and hybrid algorithm for hoist scheduling problems, in Maher, M. & Puget, J-F. (ed.), Proceedings, 4th International Conference on Principles and Practice of Constraint Programming -- CP98, Pisa, Italy, October 1998, Springer Verlag, Lecture Notes in Computer Science, 1520, 385-399

Smith, B.M. & Grant, S.A., Where the exceptionally hard problems are, Proceedings, Workshop on Studying and Solving Really Hard Problems, First International Conference on Principles and Practice of Constraint Programming, September, 1995, 172-182

Tsang, E.P.K., Foundations of constraint satisfaction, Academic Press, London and San Diego, 1993

Tsang, E.P.K., Mills, P., Williams, R., Ford, J. & Borrett, J., A computer aided constraint programming system, The First International Conference on The Practical Application of Constraint Technologies and Logic Programming (PACLP), London, April 1999, 81-93

Voudouris, C., Guided Local Search for Combinatorial Optimisation Problems, PhD Thesis, Department of Computer Science, University of Essex, Colchester, UK, 1997

Voudouris, C. & Tsang, E.P.K., Guided Local Search and its application to the Travelling Salesman Problem, European Journal of Operational Research, Anbar Publishing, Vol 113, Issue 2, March 1999, 469-499

Wallace, R.J. & Freuder, E.C., Supporting dispatchability in schedules that involve consumable resources, Journal of Scheduling, accepted for publication

Appendix A – Specification of a Sample Rostering Problem

```
/*
   The following clauses specify the senior staff, junior staff and
   assistants available (5, 16 and 8 in the following example):
*/
number_of_senior_staff( 5 ).
number_of_junior_staff( 16 ).
number_of_assistants( 8 ).


/*
   The following clauses specify the rostering demand:
      1. the number of time slots to be filled TS (=21 in this example);
      2. the number of sessions per time slot, SS (=3 in this example);
      3. the staff requirement, senior RS, junior, RJ and assistants RA
         (RS=1, RJ=3, RA=2 in this example).
*/
number_of_shifts( 21 ).
number_of_sessions( 3 ).
staff_requirements_per_slot( [senior(1), junior(3), assistant(2)] ).


/* The following clauses define the constraints */

/* Constraint 1.
   No one can work on two jobs at the same time; this applies to all
   rostering problems.
*/


/* Constraint 2.
   No staff can work for more than k consecutive sessions (k=2 in this
   example)
*/
max_consecutive_sessions( 2 ).


/* Constraint 3.
   Each staff must work for a minimum number of sessions which is no
   more than m sessions less than the average load (m=1 in this
   example)
*/
max_deviation_from_avg_load( 1 ).
```

Appendix B – A Sample Rostering Problem in EaCL

This program was generated by the **Req_to_EaCL** module of **ZDC-Rostering** system. It has been edited for improved readability.

```
Problem:Rostering
{
 Data
 {
  NS := 4;
  NJ := 6;
  NA := 5;
  NShifts := 6;
  NSessions := 2;
  MinS := 2;
  MinJ := 3;
  MinA := 3;
  RS := 1;
  RJ := 2;
  RA := 2;
  MaxCons := 3;
  MaxDev := 1;
  FirstS := 1001;
  LastS := 1004;
  FirstJ := 2001;
  LastJ := 2006;
  FirstA := 3001;
  LastA := 3005;
  NSS := 12;
  NJS := 24;
  NAS := 24;
 }
 Domains
 {
  IntDom SeniorSlots={1001, 1002, 1003, 1004};
  IntDom JuniorSlots={1001, 1002, 1003, 1004, 2001, 2002, 2003, 2004, 2005, 2006};
  IntDom AssistSlots={2001, 2002, 2003, 2004, 2005, 2006, 3001, 3002, 3003, 3004, 3005};
 }
 Variables
 {
  IntVar s[NSS]::SeniorSlots;
  IntVar j[NJS]::JuniorSlots;
  IntVar a[NAS]::AssistSlots;
 }
 Constraints
 {
  Forall (t in [0 .. NShifts - 1] )
  {
   AllDifferent([s[t*NSessions*RS+1],    s[t*NSessions*RS],    j[t*NSessions*RJ+3],    j[t*NSessions*RJ+2],
       j[t*NSessions*RJ+1], j[t*NSessions*RJ], a[t*NSessions*RA+3], a[t*NSessions*RA+2], a[t*NSessions*RA+1],
       a[t*NSessions*RA]]);
  }
  Forall (t in [0 .. NShifts - MaxCons - 1] )
```

```
   {
    Forall (sf in [FirstS .. LastS] )
    {
     Count([s[t*NSessions*RS+7], s[t*NSessions*RS+6], s[t*NSessions*RS+5], s[t*NSessions*RS+4],
         s[t*NSessions*RS+3], s[t*NSessions*RS+2], s[t*NSessions*RS+1], s[t*NSessions*RS],
         j[t*NSessions*RJ+15], j[t*NSessions*RJ+14], j[t*NSessions*RJ+13], j[t*NSessions*RJ+12],
         j[t*NSessions*RJ+11], j[t*NSessions*RJ+10], j[t*NSessions*RJ+9], j[t*NSessions*RJ+8], j[t*NSessions*RJ+7],
         j[t*NSessions*RJ+6], j[t*NSessions*RJ+5], j[t*NSessions*RJ+4], j[t*NSessions*RJ+3], j[t*NSessions*RJ+2],
         j[t*NSessions*RJ+1], j[t*NSessions*RJ]], [sf] ) <= MaxCons;
    }
    Forall (jf in [FirstJ .. LastJ] )
    {
     Count([j[t*NSessions*RJ+15], j[t*NSessions*RJ+14], j[t*NSessions*RJ+13], j[t*NSessions*RJ+12],
         j[t*NSessions*RJ+11], j[t*NSessions*RJ+10], j[t*NSessions*RJ+9], j[t*NSessions*RJ+8], j[t*NSessions*RJ+7],
         j[t*NSessions*RJ+6], j[t*NSessions*RJ+5], j[t*NSessions*RJ+4], j[t*NSessions*RJ+3], j[t*NSessions*RJ+2],
         j[t*NSessions*RJ+1], j[t*NSessions*RJ],
         a[t*NSessions*RA+15], a[t*NSessions*RA+14], a[t*NSessions*RA+13], a[t*NSessions*RA+12],
         a[t*NSessions*RA+11], a[t*NSessions*RA+10], a[t*NSessions*RA+9], a[t*NSessions*RA+8],
         a[t*NSessions*RA+7], a[t*NSessions*RA+6], a[t*NSessions*RA+5], a[t*NSessions*RA+4],
         a[t*NSessions*RA+3], a[t*NSessions*RA+2], a[t*NSessions*RA+1], a[t*NSessions*RA]], [jf] ) <= MaxCons;
    }
    Forall (af in [FirstA .. LastA] )
    {
     Count([a[t*NSessions*RA+15], a[t*NSessions*RA+14], a[t*NSessions*RA+13], a[t*NSessions*RA+12],
         a[t*NSessions*RA+11], a[t*NSessions*RA+10], a[t*NSessions*RA+9], a[t*NSessions*RA+8],
         a[t*NSessions*RA+7], a[t*NSessions*RA+6], a[t*NSessions*RA+5], a[t*NSessions*RA+4],
         a[t*NSessions*RA+3], a[t*NSessions*RA+2], a[t*NSessions*RA+1], a[t*NSessions*RA]], [af] ) <= MaxCons;
    }
   }
   Forall (sf in [FirstS .. LastS] )
   {
    Count( s, [sf] ) + Count( j, [sf] ) >= MinS;
   }
   Forall (jf in [FirstJ .. LastJ] )
   {
    Count( j, [jf] ) + Count( a, [jf] ) >= MinJ;
   }
   Forall (af in [FirstA .. LastA] )
   {
    Count( a, [af] ) >= MinA;
   }
  }
}
```