A Nurse Rostering System Using Constraint Programming and Redundant Modeling

B.M.W. Cheng, J.H.M. Lee and J.C.K. Wu
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
Email: jlee@cse.cuhk.edu.hk

Abstract

This paper describes the design and implementation of a constraint-based nurse rostering system using a redundant modeling approach. Nurse rostering is defined as the process of generating timetables for specifying the work shifts of nurses over a given period of time. This process is difficult because the human roster planner has to ensure that every rostering decision made complies with a mixture of hard hospital rules and soft nurse preference rules. Moreover, some nurse shift pre-assignments often break the regularity of wanted (or unwanted) shifts and reduce the choices for other unfilled slots. Soft constraints amount to disjunction, which can be modeled as choices in the search space. This approach, although straightforward, incurs overhead in the search of solution. To reduce search time, we propose redundant modeling, an effective way to increase constraint propagation through cooperations among different models for the same problem. Our problem domain involves around twenty-five to twenty-eight nurses and eleven shift types. Experiments and pilot testing of the system confirm the effectiveness and efficiency of our method.

1 Introduction

The problem at hand is to design and implement a nurse rostering system for the Accident and Emergency Unit of the Tang Shiu Kin Hospital, Hong Kong. The problem domain involves generating weekly timetables for a group of nurses, whose ranks, rules to be abided (be it hard or soft) and specific shift requests are different. The duty planner currently produces the duty roster manually. This task is difficult for the duty planner for three reasons. First, there are twenty-five to twenty-eight nurses of different skill classes with eleven possible shift types. The search space is enormous. The state of the art system reported in [44] can handle less than twelve nurses with three shift types. The same reported system also solves a problem with thirty nurses in a single skill class. Second, there are a number of hard hospital rules and soft nurse preference rules to be obeyed. Most of the hard rules are very tight, and many soft planning rules are often incompatible with each other. Third, the duty planner allows nurse to request shift pre-assignments. The requests often make the roster generation process even more difficult. For example, suppose a group of nurses requests to have a day-off on Sunday. If this group is too large, it may over-reduce the available manpower to work on Sunday. It may also disturb normal sequences of some wanted (or unwanted) shifts and affect the fairness and quality of the roster. In addition to assigning who to work on Sunday, the duty planner also has to consider whose shift requests to reject.

Inspired by the work of Weil *et al* [44], we employ a constraint-based approach in the construction of this nurse rostering system. This approach amounts to modeling the nurse rostering problem as a constraint satisfaction problem and using constraint programming tools for system implementation. A constraint satisfaction problem (CSP) is defined as [25]:

> We are given a set of variables, a domain of possible values for each variable, and a conjunction of constraints. Each constraint is a relation defined over a subset of the variables, limiting the combination of values that the variables in this subset can take. The goal is to find a *consistent* assignment of values to the variables so that all the constraints are satisfied simultaneously.

After modeling the problem as a CSP, constraint programming tools are used to build up the constraint-solving engine. Constraint programming tools provide a rich set of constraint-solving functions and routines, and greatly facilitates the implementation process. The expressiveness and efficiency of constraint programming in solving such kind of scheduling problem are described in [24]. In this project, we adopt ILOG SOLVER, a C++ class library [37, 17], as our tool.

We encounter two major difficulties in the development process. First, shift pre-set requests adversely affect the efficiency of the rostering system in two ways. They cause the choices for remaining shifts to become tight and limited, and break the regularity of some wanted and unwanted shifts. Second, the problem domain consists of a series of soft planning rules. We model soft rules as disjunctions in the search tree. In this approach, the search space increases combinatorially with the number of soft rules and nurses. Solution generation efficiency is degraded.

We cope with these difficulties by applying the concept of redundant modeling [6, 5] in the system. Redundant modeling is based on the observation that there is usually more than one way of modeling a problem as a CSP. By *modeling* a problem as a CSP, we mean the process of determining the variables, the associated domains of the variables, and the expressions of the constraints. Our method amounts to implementing more than one model of a given problem and *connect* the model implementations. Theoretically, not pragmatically, any one model implementation suffices to solve the CSP at hand completely. Therefore, the models are redundant with respect to one another. The combined model contains the original but redundant models as sub-models. Constraints in the sub-models cooperate during constraint-solving by propagating constraints freely amongst the sub-models. This extra level of pruning and propagation activities becomes the source of execution speedup. We create two different models for the nurse rostering problem. In the first model, the variables are nurses and the domains are the shifts; in the second model, the variables are the shifts and the domains are the nurses. The two models are then connected using channeling constraints to speed up constraint-solving [6, 5]. Experiments and pilot testing of the system confirm the feasibility of our method. In most runs, the system is able to return a quality solution in less than a minute of CPU time on a Pentium-class personal computer.

The rest of this paper is organized as follows. Section 2 provides an account of related work in the field of nursing staff rostering. In section 3, we give a brief synopsis of constraint programming and available tools. In section 4, the domain of our rostering problem is detailed. Operations and structures of the Tang Shiu Kin Hospital, as well as the various planning rules, are described in length. Section 5 describes how the technique of redundant modeling is applied in the modeling of the nurse rostering problem. We outline design and implementation issues of the rostering system in section 6. Section 7 discusses the tension between solution efficiency and quality, followed by benchmarking results in section 8. Section 9 summarizes our contributions and sheds light on future direction of research.

2 Related Work

There have been extensive attempts to solve the nurse rostering problem [38, 39, 19, 44, 21, 33, 43, 29, 28, 2], which can be classified into three main approaches. The first approach amounts to modeling the rostering problem mathematically and applying methods in operations research to search for solutions [43, 29]. This approach, though a solution can be found in a relatively short time, is too rigid to incorporate the flexibility of the problem domain. It also requires a lot of computing resources [28]. The second approach is to design domain-specific heuristics and algorithms for solving the rostering problem [33, 28, 2]. It usually formulates the problem as optimization of certain criteria, such as nurse satisfaction and staff turn-over percentage. These methods are not generic. In addition, successful algorithms and heuristics in a particular scenario may not perform well in other problem domains [24]. Moreover, these systems are often written in imperative languages, which cast large implementation overhead. The last approach employs constraint programming [44, 24, 21]. This approach is by far the most promising for its expressiveness to model problems as CSPs, in which one only needs to concentrate on the representation of variables and constraints. Solutions can be transparently generated by built-in solution generation procedure in constraint programming systems. Slight modification to the problem domain usually induces small change to the implementation. The emergence of various constraint programming tools further allow other programming methodologies, such as object-orientation, to be incorporated in constraint programming.

Our problem instance differs from those in the literature in three aspects. First, nurses in our domain prefer to work for different shifts in different days. That means that the duty planner needs to generate non-consecutive shifts patterns. This process is difficult to achieve using specific algorithms, and constraint programming facilitates expression of such requirements. On the other hand, it is relatively simple to design specific algorithms to generate consecutive shifts over consecutive days [21, 28]. Second, in most of the existing nurse rostering problems, only a few main shifts (AM, PM, Night, Day-off) are involved. However, our problem domain has a total of eleven shifts, and many of them are governed by complicated planning rules. This increases the difficulty and complexity of our problem considerably. Third, rosters are not generated as cyclic schedules, in which nurses take turn in performing each and every one of the shift patterns in the roster. There are two implications. Rosters have to be generated every week, increasing the workload of the duty planner. Fairness in shift assignment, which is one of the dominant factors that determine the morale of the nursing staff, becomes an important criterion in roster generation.

3 Constraint Programming

Real-world scheduling problems are combinatorial in nature. In its simplest form, scheduling problems are no different from classical combinatorial problems such as graph colorings, N-queen puzzles and crypt-arithmetic problem. All of the problems involve solution searching from a large search space under various kind of constraints [41]. These class of problems are collectively called constraint satisfaction problems (CSPs). In this section, we give a brief introduction to CSP, an overview of the constraint programming approach to solving CSPs, and the available tools.

3.1 Features of Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) is defined as a 3-tuple (Z, D, C), where

- Z is a finite set of variables x_1, x_2, ..., x_n,
- D is a function mapping each variable $x \in Z$ to a finite set of possible values called the *domain* of x, and
- C is a finite set of constraints, each of which acts on a subset of variables in Z restricting the possible combination of values that these variables can take.

A solution to a CSP is a *consistent* variable assignment so as to make all constraints satisfied simultaneously. A naive solution for CSP is generate-and-test but it is grossly inefficient. A more efficient approach is backtracking tree search, which essentially performs a depth-first search [20] of the space of the potential CSP solutions. The performance of backtracking, although better than generate-and-test, is still poor due to its thrashing [15] behavior.

Constraint propagation, based on node/arc/path consistency techniques [42, 25, 26, 30, 10, 3], is a more sophisticated approach to tackle CSPs. In this approach, variables are represented by nodes in a graph and binary constraints are represented by edges. During constraint propagation along the edges, values are removed from the domains of the nodes until each arc (constraint) is individually satisfiable. However, global satisfiability is not guaranteed and searching is needed. Different constraint satisfaction algorithms are obtained by combining tree searching and different degrees of constraint propagation [31, 16, 27, 14, 15, 9, 35].

Given a CSP, we first use a constraint propagation algorithm, such as the well-known arc consistency algorithm AC-3 [25] as explained above, to attain a desired level of consistency. If no solution or inconsistency is found, then one of the variables with domain size larger than 1 is selected and a new CSP is created for each possible assignment of this variable. These new CSPs become the successor of the current CSP. Repeating this constraint propagation and domain enumeration process for each new CSP results in possibly more new CSPs. All newly generated CSPs can be organized conveniently in the form of a search tree with the original CSP as the root node. A standard backtracking algorithm can be used to visit these nodes to search for solutions.

We have two choices in the above tree search process: *variable-ordering* and *value-ordering*. These orderings can affect the efficiency of the search strategies significantly. Variable-ordering concerns which variable to instantiate next; whereas value-ordering sets the order in which values in the domain of the chosen variable are tried. Variable-ordering can greatly affect the branching factor and hence also the size of the resulting search space. Value-ordering affects the ordering of branches in the search tree. This is important when users are interested in obtaining the first solution of a CSP fast using some form of depth-first search strategy. Therefore, various variable- and value-ordering heuristics [4, 16], such as the first-failed principle, have been devised to speed up solving of particular CSP instances. These heuristics are, however, usually problem and domain specific.

3.2 Constraint Programming Tools

There are a wide variety of constraint programming languages and tools, such as Prolog III [8], CLP($\mathcal{R}$) [18], and CHIP [13, 41]. These languages and tools, each differing in its constraint domain and serving in different areas of applications, offer an expressive and flexible language for problem specification and heuristics programming, allowing rapid program development for complex problems and enabling programs to be easily modified and extended. CSP-solving and scheduling are most related to CHIP and ILOG SOLVER [37], the uniqueness of which come from their finite domain constraint-solving capabilities. The idea underlying CHIP is to enhance Prolog with constraint propagation techniques [25], which prunes the search space before performing tree search. ILOG SOLVER [37] is a constraint programming language that merges constraint propagation techniques [25] and object-oriented programming. ILOG SOLVER is realized as a C++ library that embodies such constraint logic programming concepts as logical variables, incremental constraint satisfaction, and backtracking. ILOG SCHEDULE [22], built on top of ILOG SOLVER, is a library of specialized constraints for constraint-based scheduling. CHIP and ILOG SOLVER have been applied successfully to solving such industrial applications as car sequencing, disjunctive scheduling, graph coloring, and firmware design. A fuller account of their applications can be found in [11, 41, 12, 34, 36, 1, 23, 32].

4 Domain Description

Tang Shiu Kin Hospital is a multi-discipline hospital in Hong Kong. It consists of a number of operation units, the Accident and Emergency Unit (AEU) being one of them. The AEU provides 24-hour emergency services to the general public seven days a week.

Currently, there are fifty-five personnel at the AEU. Around twenty-five to twenty-eight of them are nurses of different ranks. These ranks include midwife nurses (N/M), registered nurses (RN), enrolled nurses (EN), male registered nurses (RN(m)), enrolled nurses (EN(m)) and part-time nurses (PTN).

Nurses in the AEU work in shifts. Generally speaking, there are three basic shifts in a day, namely AM shift (A), PM Shift (P), and Night Shift (N). An evening shift (E) is essentially a PM shift with a slightly different duty time. An irregular shift (I) has special working hours either arranged by the nursing officer or requested by individual nursing staff member. Nurses are entitled to different types of holiday. These includes day-off (O), compensation-off (CO), public holiday (PH) and vacation leave (VL). In addition, nurses can be pre-assigned some special work shifts such as study day (SD) and staff-on-loan (SOL). In total there are eleven shift types.

4.1 Overall Planning Operation

Around two weeks before the actual implementation of the rota, the duty planner, who is either the Ward Manager or one of the Nursing Officers, receives preferred shift requests from the staff. Based on the requests, the vacation leave list, the study day list, the Staff-on-Loan list and the past day-off history list (PH, CO, Saturday and Sunday day-off list), the duty planner generates a duty sheet and releases it to the staff. The staff can negotiate for adjustments in the duty sheet, but usually this is done through interchanging shifts with other nursing staff members.

4.2 Planning Rules

The planning rules to be respected by the duty planner can be categorized into two main types: imperative planning rules and preference planning rules. *Imperative planning rules* are rules that must be respected in any timetable. Therefore, in generating the roster, the duty planner must ensure that every planning decision made is coherent with these hard rules.

Rule 1 *Each staff member is required to work one shift per day.*

Rule 2 *Each staff member gets one O shift per week.*

Rule 3 *If a nurse is assigned an N shift on a particular day, the nurse must be assigned an A shift in the previous day.*

Rule 4 *A VL shift should start on a Monday, and should be taken in a multiple of six days. The minimum number of VL is six days. Every six VL shifts must be followed by an O shift on Sunday.*

Rule 5 *For each day, there should be eight nurses working in the A shift, six nurses working in the P shift, two nurses working in the E shift, and three nurses working in the N Shift.*

Rule 6 *There are some requirement on the number of a particular type of nurse working in a particular shift. They are summarized as follows:*

Type of shift	# Male nurses	# N/M	# EN
A	≤ 3	≥ 1	
P	≤ 3	≥ 1	
N	≤ 1	$= 1$	≤ 1

Preference planning rules should be satisfied as much as possible but they can be violated. These planning rules are mainly nurse preferences on the timetable and are results of extensive consultation and consensus with the nursing staff. Thus these rules define the "goodness" of the timetable generated. The more preference planning rules a timetable respects, the better the timetable is.

Rule 7 *A nurse should not work for the same shift for two consecutive days. In particular, a nurse prefers A shifts alternating with P/E shifts. If such an arrangement is impossible, a nurse also accepts two consecutive A or P shifts. However, assigning three consecutive A or P/E shifts to a nurse is strongly discouraged.*

Rule 8 *Apart from all holiday schedule, the number of A shifts and P shifts allocated to each nurse each week should be roughly balanced.*

Rule 9 *N shifts are unwanted. Nurses should take turn to be assigned an N shift. To be exact, if there are Y nurses of a particular rank to be scheduled for N shift, each nurse in that group should have an N shift every Y days (refer to Rule 6).*

Rule 10 *Immediately after an N shift, a nurse prefers to take an O shift. If it is not possible, a nurse prefers to take a P shift. If it is not possible, a nurse accepts an A shift.*

Rule 11 *E shift allocation is similar to Rule 9. The difference is that the AEU requires two nurses working in the E shift everyday, while three nurses are required for the N shift.*

Rule 12 *If a nurse has already performed three or four N shifts, the first compensation-off (CO1) can be allocated to the nurse. Similarly, if a nurse has already performed six or seven N shifts, the second compensation-off (CO2) can be allocated.*

Rule 13 *A nurse strongly not prefer to work for any of the following pattern.*

Undesirable Shift Patterns
P → O → P
P → O → E
E → O → P

Rule 14 *Immediately after a holiday (O, CO, PH), a nurse prefers to have a P shift.*

Rule 15 *If possible, a nurse prefers to have consecutive holidays.*

Rule 16 *A nurse prefers to work with the following patterns.*

Desirable Consecutive Patterns
A → O → P
A → N → O → P

Rule 17 *A nurse strongly prefers not to work for two consecutive P shifts on a weekend.*

Rule 18 *Each nurse should have equal opportunity to get O shifts on Saturdays and Sundays, and PH shifts on bank holidays.*

Rule 19 *Each nurse should start getting PH for a bank holiday in the period from one month before to one month after the actual date for the bank holiday.*

Rule 20 *Immediately after a VL shift, a nurse starts to work on a Monday. The nurse prefers to work in the P shift on Monday, A shift on Tuesday and N shift on Wednesday.*

5 Problem Modeling

This section describes the modeling of the system. Our system consists of two separate programs. The first program is a core constraint engine for generating rosters. This core engine is built using the principle of redundant modeling [6, 5]. This concept is to define and connect multiple, yet mutually redundant, CSP models for the same problem to form an aggregate model. The aggregate model exhibits significant speedup in constraint propagation over any of the model when used alone. The second program is a graphical user interface for user interaction. In the following, we give a brief introduction to redundant modeling, followed by a description of the two models for our nurse rostering problem. Readers interested in details of the redundant modeling approach are referred to [6, 5].

5.1 Principle of Redundant Modeling

We can view virtually any real-world scheduling process in two ways. For example, a generic job-shop scheduling problem can be formulated as assigning machines to workers or assigning workers to machines. Similarly, a school timetabling problem can be seen as allocation of courses to timeslots or allocation of timeslots to courses.

From a CSP point of view, each modeling approach corresponds to one CSP and a constraint network. There is usually not a single model that is effective enough to tackle real world problems efficiently[1]. Experience suggests that some constraints can be expressed succinctly in one model and yet the same constraints result in a cumbersome specification in another model. In addition, some constraints propagate and prune well when expressed in one model and yet expression of the same constraints in another model exhibits opposite pruning behavior.

Our approach is inspired by the effectiveness of redundant constraints[2], which are redundant semantically but augment the propagation and pruning of the original constraints of a CSP, in speeding up constraint-propagation. Intuitively speaking, our method amounts to implementing more than one model of a given problem and somehow connect the model implementations. Theoretically, not pragmatically, any one model implementation suffices to solve the CSP at hand completely. Therefore, the models are redundant with respect to one another. Each model implementation corresponds to a constraint network. In order for the different models to cooperate during constraint-solving, the network must be connected so that pruning can be propagated among the networks in a multi-directional manner. *One*

[1] An effective model of the problem should facilitate convenient expression of constraints, efficient pruning and efficient constraint-solving.

[2] Given a set of constraints C. A constraint c is *redundant* with respect to C if C entails c.

way to achieve a multi-directional connection is to use constraints to express the relationship among variables in different models. We call such constraints, which are not part of the original problem specification and exist only as an artifact of network connection, *channeling constraints*. Suppose now we have two models for the same problem. The first model assigns type X to type Y and the second model assigns type Y to type X. An effective channeling constraint in this case takes the following form:

The variable associated with object x of type X has object y of type Y as value *if and only if* the variable associated with y has x as value.

5.2 Problem Models

In this section, we describe two models for the nurse rostering problem. The first model is based naturally on the format of the roster sheet, which suggests allocation of shifts to nurses. Implementation of the first model performs well in general but fails to return answer in a timely manner for some difficult cases. This prompts us to look for another model which is complimentary to the first model in terms of constraint expressiveness and efficiency. The resulting second model is based on an allocation of nurses to shifts. Implementation of the second model again exhibits deficiency on some problem instances while performing well in general. Having two mutually redundant models in hand, we proceed to connect the models using channeling constraints.

Slot Model The slot model for the nurse rostering problem is inspired by the layout of the roster, as shown in Figure 1, which suggests allocating shifts to nurses. Each slot in

Nurse	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Nurse-01	A	E	N	P	O	SD	P
Nurse-02	P	O	A	A	CO1	PH	A
Nurse-03	SD	SD	SD	O	A	N	P

Figure 1: First View of Nurse Rostering

the roster constitutes a variable `Nurse-Day` of the model, which denotes the shift assigned to nurse `Nurse` on the day `Day`. We call each such variable a *slot variable*. The associated

Shift	Mon	Tue	Wed	Thu	Fri	Sat	Sun
A shift	N-1, N-2 N-3	N-4, N-5 N-6	N-7, N-8 N-9	N-10 N-11	N-12 N-13	N-14 N-15	N-16 N-17
P shift	N-10 N-11	N-12 N-13	N-14 N-15	N-16, N17 N-18	N-1, N-2 N-3	N-4, N-5 N-6,	N-7, N-8 N-9
O shift	N-4	N-8	N-12	N-15	N-5	N-7	N-10

Figure 2: Second View of Nurse Rostering

domains of slot variables contain the shifts available to the nurses. As the roster is generated on a weekly basis, each nurse is associated with seven slot variables.

Shift Model This model results from another view of the problem. The shifts become variables. We call them *shift variables*. Each shift variable `Shift-Day` denotes the set of nurses assigned to work in shift `Shift` on the day `Day`. There are usually more than one nurse working in a shift so that the domain of each shift variable is the powerset of the set of all nurses. The scenario is depicted in Figure 2. For a weekly roster, the number of such variables is seven times the number of shift types, and is constant regardless of the number of nurses. However, the size of the domains depends on the number of nurses.

Aggregate Model The two models are connected to form an aggregate model of the problem, using channeling constraints of the following form:

$$\texttt{Nurse-Day == Shift} \Longleftrightarrow \texttt{Nurse} \in \texttt{Shift-Day}$$

for each nurse `Nurse`, each day `Day`, and each shift `Shift`. The channeling constraints enforce that a nurse `Nurse` is assigned a shift `Shift` on day `Day` in the slot model if and only if the nurse `Nurse` is in the set of nurses assigned to shift `Shift` on day `Day` in the shift model.

5.3 Hard Constraints Modeling

Hard constraints are results of the imperative planning rules which must be satisfied at all cost. These imperative rules can be expressed conveniently as constraints since constraints are by default hard in most constraint programming tools. In the following, we discuss how we model only some of the representative constraints in the two models. Modeling of other constraints are similar. Interested readers are referred to [7].

Slot Model In the slot model, we have Rule 1 satisfied for free since the rule is implicitly satisfied by requiring each constrained variable to take on exactly one value. Rule 2 can be modeled using a counting constraint [17] as follows:

> *For each nurse* `Nurse`*, the number of variables in* {`Nurse-Mon`, ..., `Nurse-Sun`} *assigned with the O shift must be equal to one.*

Rule 3 is expressed using a boolean implication constraint "$\Rightarrow$" [17], which is a reflection of a boolean implication logical relation between two variables, as follows:

> *For each nurse* `Nurse` *and for each* `Day` *from Tuesday to Sunday,* `Nurse-Day` *is* $N \Rightarrow$ `Nurse-Day'` *is* A*, where* `Day'` = `prev(Day)`*, and* `prev()` *is a function that takes a day as input and returns the previous day of the input.*

Shift Model The variables in this model are constrained set variables [17], which have as domains a powerset. The value of a constrained set variable is thus a set. As a result, the planning rules are modeled using primitive set constraints [17] on these variables. Rule 1 is expressed using two set constraints on the eleven shifts in a day:

> *For each day* `Day`*, consider the set* V = {`A-Day`, `P-Day`, `N-Day`, ..., `SOL-Day`}*. The following two constraints must be satisfied. (1) Elements of* V *are pairwise disjoint and (2) the cardinality of* $\bigcup V$ *is equal to the number of nurses.*

Similarly, Rule 2 can be expressed as follows:

> *Consider the set* V = {`O-Mon`, ..., `O-Sun`}*. The following two constraints must be satisfied. (1) Elements of* V *are pairwise disjoint and (2) the cardinality of* $\bigcup V$ *is equal to the number of nurses.*

Since the shift variable `O-Day` denotes the set of nurses having an O shift on day `Day`, the first constraint above ensures that no nurse gets more than one O shift per week. The union of all `O-Day` shift variables for every day of a week represents the set of nurses who get O-shift in the week. Constraining the cardinality of the union ensures that every nurse gets an O shift in a week. The expression of Rule 3 in this model uses the subset constraint as follows.

Consider the sets $A = \{\texttt{A-Mon}, \ldots, \texttt{A-Sun}\}$ *and* $N = \{\texttt{N-Mon}, \ldots, \texttt{N-Sun}\}$*. For each* `Day` *from Tuesday to Sunday, we have* $\texttt{N-Day} \subseteq \texttt{A-Day'}$*, where* `Day' = prev(Day)`*, and* `prev()` *is a function that takes a day as input and returns the previous day of the input.*

5.4 Soft Constraints Modeling

Preference planning rules, or soft constraints, are expressed in the same manner as imperative planning rules. There is, however, one significant difference in how they are posted to the constraint-solving engine. For each soft constraint c, we set up a choice point. In the first branch, the constraint c is told to the solver. The other branch, one without c, is tried upon backtracking.

Some of the soft constraints are mutually incompatible. In other words, satisfying one soft constraint may make it impossible to satisfy another one. Therefore, the order by which the soft constraints are posted to the constraint engine becomes important. In the case that some soft constraints are mutually inconsistent to each other, constraints posted the last (or those closer to the leaf nodes of the search tree) are backtracked into before those posted earlier (or those closer to the root node of the search tree).

The problem at hand is to decide an order of importance among the soft constraints. There are three areas in which such ordering can be based on: (1) among each planning rule as a whole, (2) among the days for those constraints which affect a series of days, and (3) among the nurses. The first order is a user-defined parameter, which reflects the priority of different planning rules. The second and third ordering is generated randomly. This ensures that no human factor is involved in the decision process, in which fairness is an important criterion.

Rules 9 and 11 are difficult to model as constraints in the constraint-solving engine. We devise a sequential algorithm to allocate the shift for the following reasons. First, allocation of N shifts is subject to a number of difficult constraints. Second, the process involves many history records such as the date of the last N shift for each nurse, date of last E shift for each nurse, and so on. Third, there are always cases that some nurses cannot take N shift even after they have been assigned to do so because of nurse preference shift requests.

The algorithm in Figure 3 is based on the idea that the earlier the date of last shift (type

```
procedure allocate-shift(nurse[], shift-type, shift-num)
  discard nurses who cannot take up shift-type for the week
  sort nurse[] according to the date of last shift-type
  i = size(nurse[]) − shift-num
  if i < 0
    copy the first i nurses in nurse[] to the end of nurse[]
  endif
  i = 0
  day = 1
  while shift-num > 0
     current-nurse = nurse[i]
     if current-nurse is unavailable for shift-type on day
        swap current-nurse with one in assigned-nurse[]
        if not successful
           swap current-nurse with nurse[j] (j > i)
           if not successful
              swap current-nurse with a discarded nurse
              if not successful
                 replace current-nurse with one in assigned-nurse[]
                 if not successful
                    output "No shift can be allocated"
                 endif
              endif
           endif
        endif
     endif
     assign shift-type to current-nurse on day
     insert current-nurse into assigned-nurse[]
     increment(i)
     increment(day)
     decrement(shift-num)
  endwhile
endprocedure
```

Figure 3: N and E shifts Allocation Algorithm

of which can be N or E) of a nurse is, the sooner the shift should be allocated to the nurse. For example, suppose nurse X and Y have their last N shifts on November 4 and November 6 respectively. Further suppose that there are two N shifts to be allocated on November 12 and 13. It is normal to assign the N shift on the 12th to nurse X and the 13th to nurse Y. However, suppose nurse X has pre-set shift SD on November 12, causing X unable to take N shift on that day. An obvious answer is to exchange the shift with nurse Y so that nurse Y takes N shift on the 12th and nurse X takes N shift on the 13th.

Suppose there is further complication that nurse Y has also pre-set shift on the 12th. Then there must be another nurse replacing the shift. The main goal of our algorithm is to specify how to choose such nurse to replace the shift.

6 System Design and Implementation

We implement the rostering system on personal computer (PC), which is chosen over workstation for PC's wide availability. Our implementation consists of a user-interface and a rostering engine. The former is implemented in Microsoft Visual Basic 3.0 while the latter is realized in C++ with ILOG SOLVER library 3.0 [17]. Class constraints [17] of ILOG SOLVER facilitate our object-oriented design and greatly reduce our coding effort.

We show in the last section two ways of modeling the nurse rostering problem as a CSP by giving the respective sets of variables, the associated domains of the variables, and the sets of constraints. In this section, we present the system design and implementation issues of the nurse rostering system. There are five important classes of individuals in the system, namely nurses, timetable slots, shifts, days of a week, and timetables. The first two classes are used for the slot model and the remaining three classes are used for the shift model. We implement the individuals as objects in C++. In the following, we describe the definition of these object classes.

Slot class Each instance of the `Slot` class represents a slot in a timetable. It is implemented as a derived class (or subclass) of the built-in constrained integer variable [17] class of ILOG SOLVER. The principle components of a constrained integer variable are a value and an associated domain. The domain of a `Slot` object consists of the thirteen shift types[3]. Being a subclass of the constrained integer variable class, the `Slot` class inherits all the features of constrained integer variable. Therefore, we can post constraints that are applicable to constrained integer variables, such as counting constraints and arithmetic constraints, to `Slot` objects.

Nurse class Each instance of the `Nurse` object class represents a nurse in the nurse rostering system. A `Nurse` object contains two types of state information: static information and dynamic information. Static information includes:

- Name, rank and employment date of a nurse;
- Shift history records: roster of the previous week, number of N shifts taken, number of CO1 and CO2 pending, the last PH shift, dates of the last N and E shifts, dates of last CO1 and CO2 shifts, and a flag indicating whether the nurse has VL in the previous week.

This information is necessary for Rule 3, Rule 4 and Rule 5.

Dynamic information includes an eight-element array of `Slot` objects, each element of which represents a day in a week. The eighth element represents the slot on next Monday. The extra `Slot` object is necessary for the enforcement of Rules 3 and 14. In other words, each `Nurse` object holds the weekly timetable for one nurse. Accordingly, N `Nurse` objects comprise the timetable for N nurses.

In addition to state information, a `Nurse` object is associated with a number of operations. *Class constructor operations* define the initialization module of the object. These operations are executed right after object creation. Various class constraints are posted there. *State accessing functions* provide an interface to access the state information of a `Nurse` object. For example, the `getMonday` member function returns a pointer to the Monday `Slot` of the nurse. *State modifying functions* provide an interface to modify the state information of a nurse object. For example, the `setName` member function sets the name of the `Nurse` object.

[3]In our actual implementation, we introduce two additional shift types: H and UD. H is a holiday type covering any of the holiday shifts. The H type is used for tackling difficult problems. UD is a user-defined type, which is used by the duty planner to reserve the shift for special use.

Shift class Each instance of the `Shift` class denotes a shift in the problem. A `Shift` object contains the following:

- a constrained set variable, whose domain is the powerset of the set of all nurses (instances of the `Nurse` class), and
- an integer indicating to which day in a week this shift belongs.

The constrained set variable denotes the group of nurses assigned to work in that shift.

Most of the shifts in the problem can be represented by the Shift class. However, there are some special requirement constraints on the nurses in A shift, P shift, N shift and E shift. These shifts are conveniently modeled as subclasses of the `Shift` class. Consider the A shift, for example. According to Rule 5, there should be eight nurses working in the A shift. Out of these eight nurses, at most three nurses can be male, and at least one nurse should be N/M. Thus, in the `AShift` class which is a subclass of the `Shift` class, three more integer state variables are included to indicate:

- the shift size for all nurses,
- the maximum number of male nurses, and
- the minimum number of N/M nurses.

These state variables are then used by the class constructor functions of the `Ashift` class to post the corresponding constraints.

Day class Instances of the `Day` class can be viewed as a place holder of thirteen variables of `Shift` class objects. These thirteen variables denote the thirteen different shifts that a nurse can take in a day of the scheduled week. They are A, P, N, E, O, H, CO1, CO2, PH, SD, SOL, VL, UD. The `Day` class facilitates the modeling of constraints that act on all shifts within a day.

Table class There is only one instance of the `Table` class, which acts as the place holder for `Day` objects in a timetable. Currently the timetable denotes a weekly roster. Therefore seven `Day` objects are declared inside a `Table` class. The `Table` class facilitates the modeling of constraints that act on all shifts within a week.

7 Solution Efficiency Versus Solution Quality

Execution of the nurse rostering engine is divided into four steps. First, all shift pre-assignment requests are collected. Second, the N and E shifts allocation algorithm is invoked to assign E and N shifts. Third, the rest of the constraints (hard or soft) induced by the planning rules (imperative or preference) are collected and told to the constraint engine. Fourth, a standard variable labeling algorithm is used to search for solutions. There are two things to note regarding variable labeling. First, since the aggregate model contains the slot model and the shift model as sub-models, labeling variables of either model will instantiate also variables in the other model. We can choose to label either set of variables. Second, we employ the simple smallest-domain-first variable-ordering heuristics and the smallest-first value-ordering heuristics in the variable labeling process. This corresponds to the a simple application of the first-fail principle [16].

We have mentioned that the number of soft constraints satisfied is an indicator of the quality of the solution. The more soft constraints the solution satisfies, the better the solution is. Our system implements soft constraints as part of the solution searching process. This approach is by no means efficient as the search space grows exponentially to the number of soft constraints although efficiency is compensated by redundant modeling. For example, the depth of the search space increases by 6*N for implementing Rule 7 for N nurses,

In some cases, even if we do not post some of the soft constraints, the generated solution may satisfy the unposted soft constraints. The reasons are:

- Requirements of the constraints may overlap.
- The hard constraints, pre-set shift requests, other posted soft constraints affect the solution searching engine non-deterministically to produce a solution that is in favor of unposted soft constraints.

Nonetheless, these factors cannot be relied upon to enforce any soft rule. If a soft rule is important, it should be taken account in the solution search process.

To cope with this problem, we propose to parameterize the level of solution quality. A high level of solution quality requires the solution searching engine to explore more deeply into the search space by allowing more soft constraint in the search tree. A low level of solution quality requires the search engine to minimize exploration in the search space so as to generate a solution as quickly as possible. This parameter can be finely adjusted by allowing the user to specify selectively which soft constraint of which nurse to be enforced.

8 Benchmarking Results

Our real-life benchmark data contains 27 nurses and 13 shift types (2 shift types are artificially added for handling difficult cases). The theoretical complexity of the search space is 27^{13} (or 4,052,555,153,018,976,267). The rostering system produces weekly roster sheets[4]. The implementation of the slot model generates 3884 constraints, out of which 266 are imperative, during execution. Executing the shift model implementation results in 1827 constraints, out of which 1675 are imperative. These unbalanced figures reflect two phenomena. First, some types of constraints are more expressive and concise than others. Second, some planning rules that are easier to express in one model can be tedious to formulate in another model. Finally, the aggregate model consists of, in addition to the constraints from the slot and the shift models, 2079 channeling constraints.

We have tested the system based on real-life data from the hospital. The tests differ in terms of the number of shift pre-assignments and soft constraints posted. There are several ways to generate solutions from the system. First, we can search for solutions purely based on the slot model. That means solution generations amount to posting all the constraints in the slot model and instantiating the slot variables only. Alternatively, we can use the shift model, and generate rosters by instantiating the shift variables only. In both cases, the system performs satisfactorily (about 10 seconds for building up variables and the constraint network, 10 seconds to get the solutions) in the problem instances where no or few pre-set shifts are present and only one to two soft constraints are posted. When there are more pre-set shifts or more soft constraints, the system fails to generate solutions within 10 hours in some difficult cases, irrespective of the choice of models.

The other way to get rosters from the system is from the aggregate models. As mentioned before, forming an aggregate model from sub-models involves channeling constraints which

[4]Generating more than one week's roster can be achieved by running the system multiple times.

connect the variables in one model to variables in another model[5]. About 20 seconds are needed to build up the variables and constraints in this model, and again solutions can be obtained within 10 seconds. Most importantly, many cases which are unsolvable in either the slot or shift model alone can be solved efficiently using the aggregate model within 10 seconds.

In the following, we present two case studies. The first case study involves a difficult problem instance. We give and compare the timing results of the three models for the difficult problem. This study demonstrates the efficiency of the redundant modeling approach. The second case study aims at evaluating the quality of the generated roster using the percentage of soft constraints satisfied. We study the cumulative results of eleven generated rosters.

8.1 Case Study 1

Figure 4 shows the pre-set requests of a difficult week's roster, which should partially explain why this problem instance is difficult. In this week's roster assignment, 17 out of the 27 nurses request for pre-set shifts. Among them, 5 nurses request the O shift on Sunday, which is most wanted by other nurses as well. 2 nurses request VL, which extends across the entire week. Last but not least, one nurse is assigned SD from Monday to Friday and requests for O on Sunday. As a result, 47 out of 189 slots are pre-set before the roster can be generated. Some of the pre-set slots are filled in with the most undesirable (from the rostering point of view) patterns, making this week's roster exceptionally hard to generate.

The number of preference rules posted and the nature of the preference rules posted also affects the difficulty of the problem. For the same set of pre-set requests, we vary the combination of preference rules posted and show the performance of the three models in Figure 5. Column one of each row specifies the combination of preference rules imposed. Columns two and three contain the timings of the slot and shift models respectively. The string "short" means that execution finishes within 1 minute. The string "fast" means an almost instantaneous response and "long" means that the system does not return an answer within 30 mins. Column four shows the timing result of the aggregate model by labeling

[5]The aggregate model introduces more memory overhead. However, with the advent of cheaper and more massive hardware, we do not foresee this as a major obstacle

Nurse	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Nurse-01							
Nurse-02	A	O					
Nurse-03		A	P			A	
Nurse-04							
Nurse-05		P	SD	SD			O
Nurse-06							A
Nurse-07							
Nurse-08		A		A			O
Nurse-09	VL	VL	VL	VL	VL	VL	
Nurse-10					O	P	
Nurse-11							
Nurse-12						A	O
Nurse-13	SD	SD	SD	SD	SD		O
Nurse-14			O	SD		A	
Nurse-15		A					
Nurse-16	VL	VL	VL	VL	VL	VL	
Nurse-17							
Nurse-18							
Nurse-19							
Nurse-20						O	
Nurse-21		SD	A				
Nurse-22							
Nurse-23							
Nurse-24		SD					
Nurse-25						O	
Nurse-26							
Nurse-27	A		SD	SD	A		

Figure 4: A Difficult Rostering Problem

Soft Rules	Slot Model	Shift Model	Aggregate Model	
			Slot Vars	Shift Vars
—	short	short	short	short
7	long	long	1 min	2 min
7+16	long	long	fast	1 min
7+8	long	long	1 min	long
7+8+16	long	long	fast	long
7+8+16+14	long	long	fast	long
7+16+14	long	long	fast	long
7+14	long	long	long	long
7	8 min	5 min	1 min	2 min
8	1 min	long	fast	long
16	long	fast	fast	fast
12	1 min	fast	fast	fast
14	long	fast	long	fast
11	long	fast	fast	fast
15	long	long	fast	long

Figure 5: Benchmark Results for the Difficult Problem

only variables in the slot model. Alternatively, we can label only variables in the shift model and yield the results in column five. Experimental results confirm that the aggregate model exhibits significant speedup over either the slot or the shift model.

8.2 Case Study 2

Figure 6 shows the pre-set requests of a typical week's roster. The last three nurses are part-time nurses, whose shifts are all pre-set. In this week's roster assignment, 95 out of the 189 slots are pre-set in advance. Note that O pre-sets can be assigned either O, CO1, CO2, PH, or H.

In this case study, all hard constraints are posted. For soft constraints, we have tried posting them all, but yielding no solution roster within 1 CPU hour. Therefore, we relax some of the soft constraints and settle in the following combinations of soft constraints: Rules 7, 8, 9, 10, 11, 12, 13, 14, 16, and 17. An answer is returned with 10 seconds of CPU time using the aggregate model.

The first solution given by the system is shown in Figure 7. The empty slots in the figures

Nurse	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Nurse-01	N						
Nurse-02		N	N	O			A
Nurse-03	VL	VL	VL	VL	VL	VL	O
Nurse-04							N
Nurse-05	UD	O	UD	O			A
Nurse-06				N	P	A	O
Nurse-07					N	N	
Nurse-08	N						
Nurse-09	VL	VL	VL	VL	VL	VL	O
Nurse-10			P				A
Nurse-11							N
Nurse-12		P			N		
Nurse-13	UD			N			
Nurse-14		N			O		
Nurse-15						N	O
Nurse-16			A	O			A
Nurse-17		N	UD	P		A	
Nurse-18	SD	SD	SD	SD	SD	SD	O
Nurse-19					N	N	
Nurse-20	SD	SD	SD	SD	SD	SD	O
Nurse-21			N				
Nurse-22	N	A		A			
Nurse-23					O		N
Nurse-24			N	N			
Nurse-25	–	–	–	–	–	–	O
Nurse-26	–	E	–	E	–	P	O
Nurse-27	E	–	E	–	–	–	O

Figure 6: A Typical Rostering Problem

Nurse	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Nurse-01	N	A	E	P	A	O	P
Nurse-02	A	N	N	O	P	A	A
Nurse-03	VL	VL	VL	VL	VL	VL	O
Nurse-04	E	P	A	O	P	A	N
Nurse-05		O		H	A	P	A
Nurse-06	P	CO1	A	N	P	A	O
Nurse-07	A	O	P	A	N	N	P
Nurse-08	N	E	PH	P	A	O	P
Nurse-09	VL	VL	VL	VL	VL	VL	O
Nurse-10	A	O	P	A	E	P	A
Nurse-11	A	O	P	A	E	A	N
Nurse-12	CO2	P	P	A	N	O	P
Nurse-13		P	A	N	A	E	O
Nurse-14	A	N	A	E	O	P	A
Nurse-15	P	A	PH	P	A	N	O
Nurse-16	P	P	A	O	P	PH	A
Nurse-17	A	N		P	O	A	E
Nurse-18	SD	SD	SD	SD	SD	SD	O
Nurse-19	O	P	A	A	N	N	E
Nurse-20	SD	SD	SD	SD	SD	SD	O
Nurse-21	P	A	N	O	P	E	PH
Nurse-22	N	A	P	A	A	O	P
Nurse-23	P	A	O	P	H	A	N
Nurse-24	CO2	A	N	N	O	P	A
Nurse-25							O
Nurse-26		E		E		P	O
Nurse-27	E		E				O

Figure 7: A Typical Generated Roster

are reserved shifts that are marked either as "–" or "UD" in the pre-set process. The duty planner has the discretion of assigning special shifts to these slots.

In the following, we measure the quality of the generated roster by the percentage of soft constraints satisfied in the roster. Figure 8 shows the degree of satisfaction of soft constraints in the generated roster. The percentage for each rule represents the ratio of the number of

Rule	Satisfied	Percentage
7	27	100%
8	24	88%
9	27	100%
10	12	44%
11	27	100%
12	27	100%
13	26	96.2%
14	18	54.5%
15	0	0%
16	14	51.8%
17	27	100%

Figure 8: Soft Constraint Satisfaction of a Single Week

satisfied soft constraints to the total number of soft constraints generated from the rule. Consider Rule 14. This rule requires that all holidays should be followed by a P or an E shift. The total number of possible soft constraint is 35, as there are totally 35 holidays allocated. Out of these 35 holidays, only 18 of them have P/E shift following them. This gives a soft constraint satisfaction of 54.5%. Other percentages are calculated similarly.

We perform ten further experiments on the system. The results obtained, recorded in Figure 9, are in line with the results in Figure 8.

9 Concluding Remarks

During our implementation, the generated rosters are sent to the hospital duty officer for evaluation. He compares the rosters with his manually generated counterparts, and returns comments to improve the quality of the solutions. We follow his advise and improve upon the system. This iterative process lasts for about three months, and now the system is ready

Rule	Satisfied	Percentage
7	254	98%
8	240	93%
9	257	100%
10	120	46%
11	257	100%
12	257	100%
13	249	96.8%
14	194	64.6%
15	0	0%
16	121	47%
17	257	100%

Figure 9: Soft Constraint Satisfaction of Ten Weeks

for pilot run. The contribution of this project is three-fold. We have successfully completed the construction of a real-life nurse rostering system, which consists of a constraint-solving engine and an ergonomic user-interface. The system is able to produce rosters for twenty-seven nurses of different skill classes with eleven shifts in a timely manner. The generated rosters are also of high quality. Our system compares well against the state of the art. It demonstrates the potential of constraint technology in the computerization of hospital management. Instead of consuming time in generating the rosters, hospital officers can spare more time and effort on patients and other medical duties. Other areas of hospital management, such as nurse duty assignment, patient-ward allocation, should also benefit from constraint technology.

In the development process of the core engine, we observe that techniques in solving classical CSPs alone are often insufficient to cope with real-life problems. In particular, soft constraint is not part of a standard CSP, and yet it is an important element in our problem. We adopt to model soft constraints as choices in the solution search tree, but this leads to a considerably large search space, thus increasing solution search time. We alleviate this problem by applying a novel technique, redundant modeling. Benchmarking results confirm substantial improvement in solution generation efficiency.

A future direction of development is to design alternative methods for implementing soft constraints in the constraint-solving engine where no notion of soft constraints is provided.

These methods include designing variable- and value-ordering heuristics and optimizing objective functions which reflect the quality of solution. Hierarchical and stochastic constraint methods can also be investigated for their suitability to cope with soft constraints.

Van Emden *et al* [40] suggest that system users do not usually know exactly what they want, and yet afterwards they do know what they do not want. This observation holds true in our implementation experience in this project. When we first meet with the duty planner, he gives us a set of planning rules. In subsequent meetings, he often discovers that he either misses or mistakes some details, thus leading to changes to the old planning rules or even creation of new planning rules. Yet we are able to cope with these frequent changes and modify our implementation accordingly. We attribute our flexibility to the expressiveness of constraint programming. Changing the problem specifications, in most cases, simply implies modifying the domains of the variables and expression of the associated constraints. Such high level of flexibility is difficult to achieve using mathematical programming or problem-specific algorithms.

Acknowledgement

We are indebted to Mr. Vincent Tam of the Tang Shiu Kin Hospital, who spared great patience in explaining to us the planning rules and rostering techniques. Mr. Tam is also always available to answer our queries. Constructive comments from the anonymous referees of the Eighth IEEE International Conference on Tools with Artificial Intelligence and the Second International Conference on Principles and Practice of Constraint Programming help improve the quality of the final version of the paper.

References

[1] P. Baptiste and C. Le Pape. Disjunctive constraints for manufacturing scheduling: Principles and extensions. In *Proceedings of the Third International Conference on Computer Integrated Manufacturing*, Singapore, 1995.

[2] P.C. Bell, G. Hay, and Y. Liang. A visual interactive decision support system for workforce (nurse) scheduling. *INFOR*, 24(2):134–145, 1986.

[3] C. Bessière and M. Cordier. Arc-consistency and arc-consistency again. In *Proceedings of AAAI-93*, pages 108–113, 1993.

[4] J. Bitner and E.M. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18:651–655, 1985.

[5] B.M.W. Cheng, J.H.M. Lee, and J.C.K. Wu. A constraint-based nurse rostering system using a redundant modeling approach. In *(to appear) Proceedings of the Eighth IEEE International Conference on Tools with Artificial Intelligence*, November 1996.

[6] B.M.W. Cheng, J.H.M. Lee, and J.C.K. Wu. Speeding up constraint propagation by redundant modeling. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 91–103, 1996.

[7] B.M.W. Cheng and J.C.K. Wu. Nursing staff rostering using ILOG solver. Final year project report, Department of Computer Science and Engineering, The Chinese University of Hong Kong, 1996.

[8] A. Colmerauer. An introduction to Prolog III. *Communications of the ACM*, pages 69–90, July 1990.

[9] R. Dechter and I. Meiri. Experimental evaluation of preprocessing techniques in constraint-satisfaction problems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 37–42, Menlo Park, Calif., 1989. American Association for Artificial Ingelligence.

[10] Y. Deville and P. Van Hentenryck. An efficient arc consistency algorithm for a class of CSP algorithm. In *Proceedings of IJCAI 1991*, pages 325–330, 1991.

[11] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving large combinatorial problems in logic programming. *Journal of Logic Programming*, 8:75–93, 1990.

[12] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, and T. Graf. Applications of CHIP to industrial and engineering problems. In *Proceedings of the 1st International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems*, pages 887–892, 1988.

[13] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, Japan, December 1988.

[14] J. Gaschnig. Experimental case studies of backtrack versus Waltz-type versus new algorithms for satisfying assignment problems. In *Proceedings of the Second Biennial Conference of the Canadian Society for Computational Studies of Intelligence*. Canadian Information Processing Society, 1978.

[15] J. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, U.S.A., 1979.

[16] R.M. Haralick and G.L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[17] ILOG. *ILOG: Solver Reference Manual Version 3.0*, 1995.

[18] J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[19] M.M. Kostreva and K.S.B. Jennings. Nurse scheduling on a microcomputer. *Computers Operations Research*, 18(8):731–739, 1991.

[20] V. Kumar. Depth-first search. In S.C. Shapiro, editor, *Encyclopaedia of Artificial Intelligence*, volume 2, pages 1004–1005. Wiley, 1987.

[21] Jose Mari Lazaro and Pedro Aristondo. Using SOLVER for nurse scheduling. In *Proceedings of ILOG SOLVER & ILOG SCHEDULE First international users' conference*, July 1995.

[22] C. Le Pape. Implementation of resource constraints in ILOG SCHEDULE: A library for the development of constraint-based scheduling systems. *Intelligent Systems Engineering*, 3:55–66, 1994.

[23] C. Le Pape. Resource constraints in a library for constraint-based scheduling. In *Proceedings of the INRIA/IEEE Conference on Emerging Technologies and Factory Automation*, Paris, France, 1995.

[24] C. Le Pape. Using object-oriented constraint programming tools to implement flexible 'easy-to-use' scheduling systems. In *Proceedings of the NSF Workshop on Intelligent, Dynamaic Scheduling for Manufacturing*, January 1993.

[25] A.K. Mackworth. Consistency in networks of relations. *AI Journal*, 8(1):99–118, 1977.

[26] A.K. Mackworth, J.A. Mulder, and W.S. Havens. Hierarchical arc consistency: Exploiting structured domains in constraint satisfaction problems. *Computational Intelligence*, 1:118–126, 1985.

[27] J. McGregor. Relational consistency algorithms and their applications in finding subgraph and graph isomorphism. *Information Science*, 19:229–250, 1979.

[28] J.D. Megeath. Successful hospital personnel scheduling. *Interfaces*, 8(2):55–59, 1978.

[29] H.E. Miller. Nurse scheduling using mathematical programming. *Operations Research*, 24(8):857–870, 1976.

[30] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[31] B. Nadel. Tree search and arc consistency in constraint-satisfaction algorithms. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 287–342. Springer-Verlag, 1988.

[32] W. Nuijten and E. Aarts. A computational study of constraint satisfaction for multiple capacitated job shop scheduling. *European Journal of Operational Research*, 1995.

[33] I. Ozkarahan and J.E. Bailey. Goal programming model subsystem of a flexible nurse scheduling support system. *IIE Transactions*, 20(3):306–316, 1988.

[34] M. Perrett. Using constraint logic programming techniques in container port planning. *ICL Technical Journal*, 7(3):537–545, May 1991.

[35] P. Prosser. Hybrid algorithms for the constraint-satisfaction problem. Research Report AISL-46-91, Computer Science Department, Univ. of Strathclyde, 1991.

[36] J.-F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *Proceedings of 7th ISMIS*, Trondheim, Norway, June 1993.

[37] J.-F. Puget. A C++ implementation of CLP. In *Proceedings of SPICIS 94*, November 1994.

[38] S.U. Randhawa and D. Sitompul. A heuristic-based computerized nurse scheduling system. *Computers Operations Research*, 20(8):837–844, 1993.

[39] S.P. Siferd. Workforce staffing and scheduling: Hospital nursing specific models. *European Journal of Operational Research*, 20:233–246, 1992.

[40] M.H. van Emden, M. Ohki, and A. Takeuchi. Spreadsheets with incremental queries as a user interface for logic programming. *New Generation Computing, OHMSHA, LTD. and Springer-Verlag*, 4:287–304, 1986.

[41] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.

[42] D. Waltz. Understanding line drawings of scenes with shadows. In P.H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.

[43] D.M. Warner. Scheduling nursing personnel according to nursing preference: A mathematical programming approach. *Operations Research*, 24(8):842–856, 1976.

[44] G. Weil, K. Heus, P. Francois, and M. Poujade. Constraint programming for nurse scheduling. *IEEE Engineering in Medicine and Biology*, 14(4):417–422, July/August 1995.