Available online at www.sciencedirect.com
ScienceDirect
Knowledge-Based Systems 20 (2007) 160–169
Knowledge-Based SYSTEMS
www.elsevier.com/locate/knosys

Managing restaurant tables using constraints

Alfio Vidotto [a,*], Kenneth N. Brown [a], J. Christopher Beck [b]

[a] Cork Constraint Computation Centre, Department of Computer Science, University College Cork, 14 Washington Street West, Cork City, Ireland
[b] Toronto Intelligent Decision Engineering Laboratory, Department of Mechanical and Industrial Engineering, University of Toronto, Canada

Received 11 October 2006; accepted 16 November 2006
Available online 8 December 2006

Abstract

Restaurant table management can have significant impact on both profitability and the customer experience. The core of the issue is a complex dynamic combinatorial problem. We show how to model the problem as constraint satisfaction, with extensions which generate flexible seating plans and which maintain stability when changes occur. We describe an implemented system which provides advice to users in real time. The system is currently being evaluated in a restaurant environment.

Keywords: Constraints; Changes; Uncertainty; Restaurant management

1. Introduction

Effective table management can be crucial to a restaurant's profitability – inefficient use of tables means that the restaurant is losing potential custom, but overbooking means that customers are delayed or feel cramped and pressured, and so are unlikely to return. In addition, customer behaviour is uncertain, and so seating plans should be flexible or quickly reconfigurable, to avoid delays. The restaurant manager is faced with a series of questions. Should a party of two be offered the last four-seater table? For how long should we keep a favourite table for a regular customer? Should a party of four be offered a table for 8 p.m.? If no table is available at 7 p.m., what other times should be offered? When a party takes longer than expected, can we re-assign all diners who have not yet been seated to avoid delays? When a party does not appear, can we re-assign all other diners to gain an extra seating? In Computer Science terms, table management is an online constrained combinatorial optimisation problem – the restaurant must manage reservations, and manage unexpected events in real-time, while maximising the use of its resources.

In this paper, we describe an implemented solution to the restaurant table management problem which helps managers to answer the above questions. The solution is based on constraint programming, and handles both flexibility and stability. The system we describe is currently being evaluated in a restaurant. The remainder of the paper is organised as follows. Section 2 presents more details of the table management problem, and describes one particular restaurant. Section 3 reviews the necessary elements of constraint programming. Section 4 presents a basic constraint model and search algorithm. Section 5 extends the model to represent flexibility, and to search for flexible plans, while Section 6 describes our approach to finding stable plans. Section 7 presents the user interface for our implemented system. Finally, Section 8 describes conclusions and future work.

2. Restaurant table management

Eco [1] is a popular medium-size restaurant in Douglas, Cork City, with a high turnover seven days a week. It was a pioneer in computer and Internet solutions, first offering email booking in 2000. The restaurant has 23 tables,

* Corresponding author. Tel.: +353 21 4904444; fax: +353 21 4255424.
E-mail addresses: av1@student.cs.ucc.ie (A. Vidotto), k.brown@cs.ucc.ie (K.N. Brown), jcb@mie.utoronto.ca (J.C. Beck).

0950-7051/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.knosys.2006.11.002

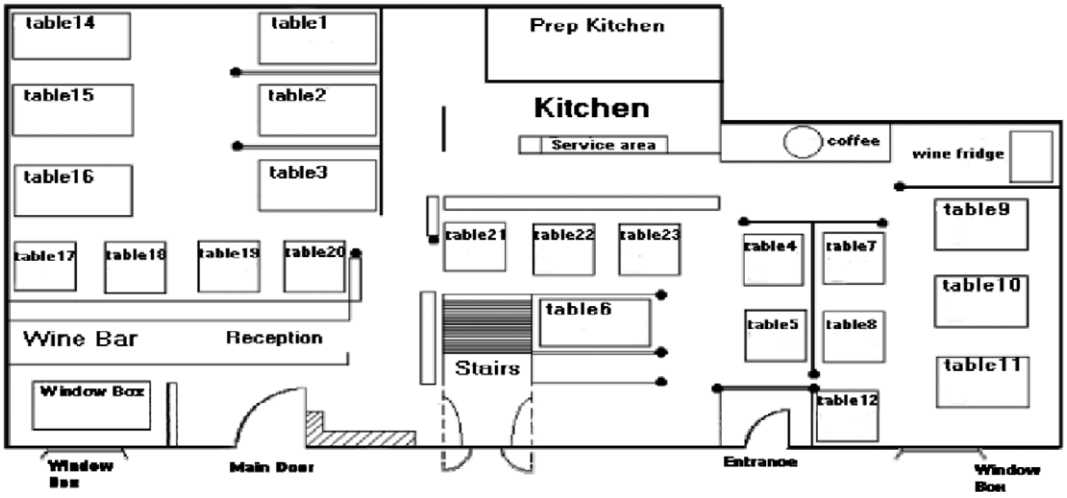

Fig. 1. Layout of the restaurant *Eco*.

ranging in size from 2 to 8 (Fig. 1). Some of the table capacities depend on the state of other tables: for example, tables 2 and 15 can both seat 6, but when one is occupied by 5 or 6 diners, then the other can seat at most 4. The tables can also be reconfigured: for example, the 2-seater tables 21 and 22 can be joined to accommodate 3–5 diners. The maximum party size that can be seated at a conjoined table is 30. There are over 100 different possible restaurant configurations, and thus the restaurant capacity ranges from 85 to 94. An evening session in the restaurant begins at 4 p.m., and the last party should be seated by 10:30 p.m. As a guide, the restaurant aims to have between 190 and 210 covers (individual diners) each evening – fewer than that, and the tables are not being well utilised; more than that, and the kitchen will be stretched to provide the food on time. Table management in Eco, as in most restaurants, has two distinct phases: *booking* and *floor management*.

In the booking phase, the booker must negotiate start times with customers to ensure that customers' requirements are satisfied, while maintaining a flexible table assignment that maximises the chances of being able to seat the desired number of covers. Typically, the booker will allocate specific tables to each booking request, and these rarely change; when a request cannot be accommodated on the current booking sheet, either the customer must be persuaded to accept another time, or the request must be declined. It is possible, however, that a reallocation of diners to tables would allow the new request to be accepted. In some cases, in order to maintain a balanced plan, a restaurant will decline a booking, or suggest a different time, even if a table is available. In addition, the booker must estimate the expected duration of the meal, based on the characteristics of the booking (including time, day of the week, and party size).

In floor management, the objectives are different. The evening starts with a partially completed booking sheet. The customers have been given definite times, and the aim is now to seat the customers with minimum delay, to modify the seating plan when changes happen, and to accept or decline "walk-ins" – customers arriving at the restaurant without a booking. The main challenge is that individual customers are unpredictable – they may arrive late, they may not arrive at all, they may take longer or shorter than expected, they may change the size of their party, and they may arrive believing a booking has been made when none has been recorded. The floor manager must make instant decisions, balancing current customer satisfaction with expectations for the rest of the evening.

The initial problem is to construct an interactive software tool, which assists restaurant staff in both the booking and floor management phases. As a research problem, our goal is to evaluate whether constraint programming techniques can provide support for the dynamic and uncertain aspects of the problem. If the research prototype is successful, a new tool will be developed, and incorporated into customer relationship management software.

3. Constraint programming

A *Constraint satisfaction problem* (CSP) is defined by a set of decision variables, $\{X_1, X_2, \ldots, X_n\}$, with corresponding domains of values $\{D_1, D_2, \ldots, D_n\}$, and a set of constraints, $\{C_1, C_2, \ldots, C_m\}$. Each constraint is defined by a scope, i.e. a subset of the variables, and a relation which defines the allowed tuples of values for the scope. A state is an assignment of values to some or all of the variables, $\{X_i = v_i,\ X_j = v_j, \ldots\}$. A solution to a CSP is a complete and consistent assignment, i.e. an assignment of values to all of the variables, $\{X_1 = v_1,\ X_2 = v_2,\ \ldots,\ X_n = v_n\}$, that satisfies all the constraints. The standard methods for solving CSPs are based on backtracking search interleaved with constraint propagation. An introduction to constraint programming can be found in [2], while [3] surveys recent research.

For search, the order in which variables and values are tried has to be specified as part of the search algorithm, and has a significant effect on the size of the search tree. The standard variable ordering heuristic chooses the

variable with the smallest current domain, or the smallest ratio of domain size to the number of constraints acting on the variable. For an instance of a CSP, a single run with a single ordering heuristic can get trapped in the wrong area of the search tree. To avoid this, randomised restarts have been proposed [4] – for a single heuristic, if no result has been found by a given time limit, the search is started again. Tie breaking and value ordering are done randomly, and so each restart explores a different path. Similarly, algorithm portfolios [5] interleave a set of randomised algorithms. In [6] search robustness is enhanced by combining multiple variable and value ordering heuristics with time-bounded restarts.

In constraint propagation, the domains of unassigned variables are reduced by removing values which cannot appear in any solution that extends the current state. For example, if we have the constraint $X < Y$, and X and Y's domains are $\{2,3,4,5\}$ and $\{1,2,3,4\}$, respectively, then the values 4 and 5 can be removed from X's domain, and 1 and 2 from Y's domain, since none of those values could possibly satisfy the constraint. Reducing the domains reduces the size of sub-tree that has to be explored. A large part of the success of constraint programming tools is due to efficient domain filtering algorithms for specialised constraints; e.g. [7].

Dynamic problems are problems that change as the solution is being executed – for example, in scheduling, a machine may break down, or a scheduled action may be delayed due to the late arrival of supplies. Dynamic CSPs [8] model changes to problems. The aim may be to minimise the effort to find new solutions, or to minimise the distance between successive solutions. Attention has recently turned to problems where we have some model of what the changes might be. Both [9] and [10] reason about the probability of future events: [9] searches and propagates constraints over a tree of possible futures; [10] samples possible futures, and then selects an action which minimises regret over the samples. [11] searches for optimally stable solutions. They start with the original solution and iteratively check whether reassigning one variable, two variables, etc., is sufficient to solve the new problem. [12] proposes special stability constraints. Some approaches aim to prevent instability by providing robust solutions. In [13] flexible solutions to scheduling problems are achieved by adding slack to activity durations. Super solutions [14] are solutions that guarantee a limited number of repairs in case of changes.

4. Modelling the static table management problem

As discussed in Section 2, the restaurant problem is inherently dynamic, but we can view it a sequence of static problems, each linked by a set of changes. In this section, we describe our representation of the static problem as a CSP, and discuss our algorithm for solving it.

We model table management as a scheduling problem, viewing tables as resources, and parties as tasks. Each party has a fixed start and end time, and a size. Each party must then be allocated to a table (or set of tables), such that the table is large enough for the party, and such that no two parties that overlap in time are allocated to the same table. Each party must be seated without interruption on the table. The problem is to determine whether or not a set of parties can be seated, and to provide a feasible seating plan if there is one. Despite having fixed start and end times, the underlying scheduling problem is NP-complete [15]. Fig. 2 shows a problem instance with five parties (left) and a possible allocation (right), where tables T_2 and T_3 have been joined for the first two time slots.

To represent this as a CSP, we model the parties as decision variables, and the tables as the values to be assigned. The detailed constraint model is generated automatically

Party	Size	Start	End
P_1	2	0	2
P_2	4	0	2
P_3	3	1	3
P_4	2	2	4
P_5	2	2	4

Table[size]	0	1	2	3
T_1[2]	P1		P4	
T_2[3]	P2		P5	
T_3[3]				
T_4[4]		P3		

Fig. 2. Problem instantiation at time 0 (left); and a possible seating plan (right).

Variables: $\{P_1, P_2, P_3, P_4, P_5\}$
Domains: $D_1=\{T_1, T_2, T_3, T_4\}$, $D_2=\{T_2, T_4\}$, $D_3=\{T_2, T_3, T_4\}$, $D_4=\{T_1, T_2, T_3, T_4\}$, $D_5=\{T_1, T_2, T_3, T_4\}$
Constraints:

C_1. alldifferent($[P_1,P_2,P_3]$)
C_2. alldifferent($[P_3,P_4,P_5]$)
C_3. $(P_2==T_2) \Rightarrow (P_1 \neq T_3, P_3 \neq T_3)$
C_4. $(P_3 \neq T_3) \| (P_2 \neq T_4)$
C_5. $3 + (P_2==T_2) \leq 4$
C_6. $P_1.size + P_2.size + P_3.size \leq 12$
C_7. $P_3.size + P_4.size + P_5.size \leq 12$
C_8. $P_4 < P_5$

Fig. 3. CSP model for the problem of Fig. 2.

from a template and from details of the restaurant. Fig. 3 shows the resulting model for the simple problem of Fig. 2. The variables P_1, P_2, P_3, P_4, and P_5 can take values from the domains D_1, D_2, D_3, D_4, and D_5 respectively. Since T_3 can be joined onto T_2 to give a capacity of 6, T_2 appears in D_2. Constraints C_1 and C_2 ensure that any parties overlapping in time use different tables. C_3 ensures that if the extra capacity of T_2 is required, then T_3 cannot be assigned simultaneously (P_2 is the only party that could require the increased capacity). C_4 is an extra constraint that ensures that T_3 and T_4 cannot both be fully occupied at the same time (which could only happen if they are assigned P_3 and P_2 respectively). C_5 ensures that in timeslot 1, the number of usable tables is not less than the number of parties, where the number of usable tables is decremented each time two tables are joined. C_6 and C_7 similarly ensure that the number of seats is not less than the number of diners. For this example, C_5, C_6 and C_7 are always true, but are shown here for illustration. Finally, C_8 breaks a symmetry in the problem, and ensures that an ordering is forced between pairs of equivalent parties.

Restaurant table management is a real-time problem – neither the booker nor the floor manager can wait for an exhaustive search before replying to a customer. Therefore, we impose a time limit on each search, and if no seating plan is found within that limit, we report no solution. Even with the time limit, though, solvers can give widely varying results depending on the particular search heuristic used. Initial tests showed that search based on a single heuristic may solve some instances quickly, but can be too slow on others, exceeding the time limit. Different heuristics tried over the same set of instances showed different partitions between hard and easy instances. However, there were very few instances that none of the heuristics could solve.

Therefore, we devised a restart approach with multiple different ordering heuristics, and an increasing time limit for each set of restarts. This *multi-heuristic* algorithm (MH) was described in [6], where we demonstrated the benefit, in terms of efficiency and robustness, of the approach. The pseudocode for the algorithm is shown below.

```
while Solve(heuristic(i),limit) == false
  limit = Increase(i,limit)
  if i == n then i = 1
  else i = i + 1
```

Solve(.,.) takes heuristic i (composed of a variable and a value ordering), and applies standard search up to a time *limit*. If it finds a solution, or proves there is no solution, it returns *true*; otherwise it hits the time limit and returns *false*. *Increase*(.,.) is the time limit function and takes the form $Increase\ (i, limit) = limit{*}10\ \ if\ i = n;\ limit\ otherwise$. MH thus tries each ordering in turn for a limited time, restarting the search after each one, and gradually increasing the time limit if no result was found. This is similar to the way iterative deepening [16] explores each branch to a certain depth, and then increases the depth limit, and is similar to randomised restarts, except we use different ordering heuristics. In total, we have 11 different variable ordering heuristics, including versions of min-size-domain and lexicographic, and including orders based on increasing and decreasing party size and start time. We have 3 different value orderings (increasing table size, decreasing table size, and lexicographic), giving a total of 33 different heuristic combinations.

Using this model configuration we are able to solve the static problem efficiently. Instances representing a full booking sheet of 200 covers can be solved in less than 0.5 s on average (examples will be shown in Section 7). Note that the real problems are typically smaller than this, either because we build the plan incrementally, or when we react to changes, some diners have already started and cannot be moved.

5. Flexibility and optimisation

The previous section described a satisfaction problem: i.e. it does not consider optimisation, but simply returns the first allocation it finds, or reports failure. However, there are likely to be many possible seating plans, and some will be significantly better than others in terms of efficient use of the tables, and thus in their ability to accept future bookings. In this section we describe a measure to estimate the quality of a solution, and an algorithm which uses that measure to search for seating plans of increasing quality.

Ultimately, seating plans should be assessed by the final number of covers achieved. Therefore, whether we are in the booking phase or in the floor management phase, we should maintain a seating plan aimed at maximising the covers. Thus after each change, we should be searching for:

$$\mathrm{argmax}_{\text{seating plan}}[\text{current covers} + \text{expected future covers}] \tag{1}$$

As the number of *current covers* is known and constant, we focus on the *expected future covers*. We do not have well-founded distributions of the new requests we can expect, and so our measure of expected covers must be an approximation. Thus we introduce a heuristic measure, *flexibility*, and search for:

$$\mathrm{argmax}_{\text{seating plan}}[flexibility] \tag{2}$$

The flexibility measure is based on the number of usable start times for future requests. Let TB be the number of tables, and T be the time horizon discretised in 15-min units. We superimpose a grid G of size $\mathrm{TB} \times T$ over the seating plan. For each grid square (table, time-unit) in G that corresponds to an unoccupied slot we compute the number of time units available before the table becomes occupied again. Squares with numbers less then a standard dinner duration d are ignored, as they do not represent usable start times. We then compute flexibility as follows:

$$flexibility = \Sigma_{tb \in TB, tu \in T}((G[tb, tu] \geqslant d) \times size(tb)) \tag{3}$$

	0	1	2	3	4	5	6	7	8
$T_1[2]$	1		$P_1[3]$			**4**	**3**	2	1
$T_2[2]$	1					**4**	**3**	2	1
$T_3[3]$	**8**	**7**	**6**	**5**		**4**	**3**	2	1

(i)

	0	1	2	3	4	5	6	7	8
$T_1[2]$	**8**	**7**	**6**	**5**		**4**	**3**	2	1
$T_2[2]$	**8**	**7**	**6**	**5**		**4**	**3**	2	1
$T_3[3]$	1		$P_1[3]$			**4**	**3**	2	1

(ii)

Fig. 4. Flexibility maps for two possible allocations.

The term $(G[tb, tu] \geqslant d)$ takes value 1 when the pair (tb, tu) represents a *usable start time*, and 0 otherwise, while $size(tb)$ is the size of table (tb).

As an illustration, consider Fig. 4, which shows a restaurant with 3 tables: T_1 and T_2 have capacity 2, T_3 has capacity 3, and T_1 and T_2 can be joined to give a capacity of 4. The evening is divided into 8 time units. Party P_1 (size 3, start 1, end 4) has two possible allocations, shown in (*i*) and (*ii*). The remaining grid cells show the number of time units available. If we assume the standard dinner duration is $d = 3$, then we count only squares with value at least 3, and we obtain: *flexibility* $(i) = (2 \times 2) + (2 \times 2) + (3 \times 6) = 26$, and *flexibility* $(ii) = (2 \times 6) + (2 \times 6) + (3 \times 2) = 30$, and thus plan (ii) would be preferred. Note that the values for T_3 are given a higher weight, since it can seat more customers.

For each problem instance, we then perform a branch-and-bound search, optimising for flexibility. Inside the search, we again apply the multi-heuristic approach. The benefit resulting from applying this optimisation criterion is illustrated in Section 7 (Figs. 9, 10 and 12).

6. Minimising disruption

The constraint satisfaction and optimisation models described above do not consider the number of table reallocations from one plan to the next – their aim is to find any (improving) plan. During the floor management phase, however, too many changes causes confusion in the restaurant, making it difficult for staff to understand and evaluate each new solution. In particular, frequent changes in the table configurations will annoy both staff and customers. Therefore, the table management system should, when possible, try to maintain the stability of the plan, and should prefer new plans with few changes.

Therefore we extend the previous models, so that when changes occur, we search for new solutions in two phases: first, we search in the neighbourhood of the previous solution, placing a limit on the number of changes allowed; second, if no acceptable plan is found in the first phase, we allow all allocations to float, and we search for any new solution. The pseudocode is shown below.

```
solution = original
discrepancy = 0
while ((timer < timeout_1) && (discrepancy < discrepancyMax))
  if Solver.solve(CSP,MH,timeout_1,original, discrepancy) == true
    solution = getSolution()
    return solution
  else discrepancy += 1
if  Solver.solve(CSP,MH,timeout_2,original, any) == true
  solution = getSolution()
return solution
```

The number of allowed changes from the *original* solution is represented by the variable *discrepancy*. The initial *discrepancy* limit is set to 0: i.e. we first check to see if the new event can be integrated into the original solution without any further changes. If not, the discrepancy limit is incremented until either a solution is found, or the limit reaches *discrepancyMax*. In the latter case, a final search is carried out for a new solution with no limit on the number of changes. The solve procedure is extended to include the discrepancy limit, which is posted as a constraint on the solution. A similar procedure is applied when searching for flexible solutions, which allows the user to trade-off stability for flexibility. Section 7 (Fig. 12) will illustrate how this is performed.

7. The integrated table management adviser

The models and algorithms described in the previous three sections have been implemented using Ilog Solver 6.0. Access to the models is provided by a graphical user interface, which also presents other relevant information regarding the state of the restaurant or booking sheet, and allows the user (booker or floor manager) to control the table allocation process, switching between manual operation, basic solving, optimising for flexibility, or maintaining stability.

A screen shot of the interface is shown in Fig. 5, displaying one possible seating plan on one evening in May 2006. The list on the left side displays in alphabetical order the parties (with time, name and table) which are allocated on the plan. New booking requests are processed by editing a form, and selecting time, party size, and expected duration. The user has the option to specify or forbid a table for the new party; otherwise the system will use any suitable table.

Fig. 6 represents the seating plan accommodating the new request (Keane). It also shows the total covers, the covers partitioned in 3 periods, the total parties, the number of parties seated at oversized tables, and the number of changes from the previous plan. Note that O'Grady at 5:30, Buckley at 6:00, O'Driscoll at 7:00 and Counihan at 9:30 are all seated at conjoined tables.

By default, the system does not allocate parties of 2 into four-seater or larger tables, but the user can override this and specify a preference for a more comfortable table. In

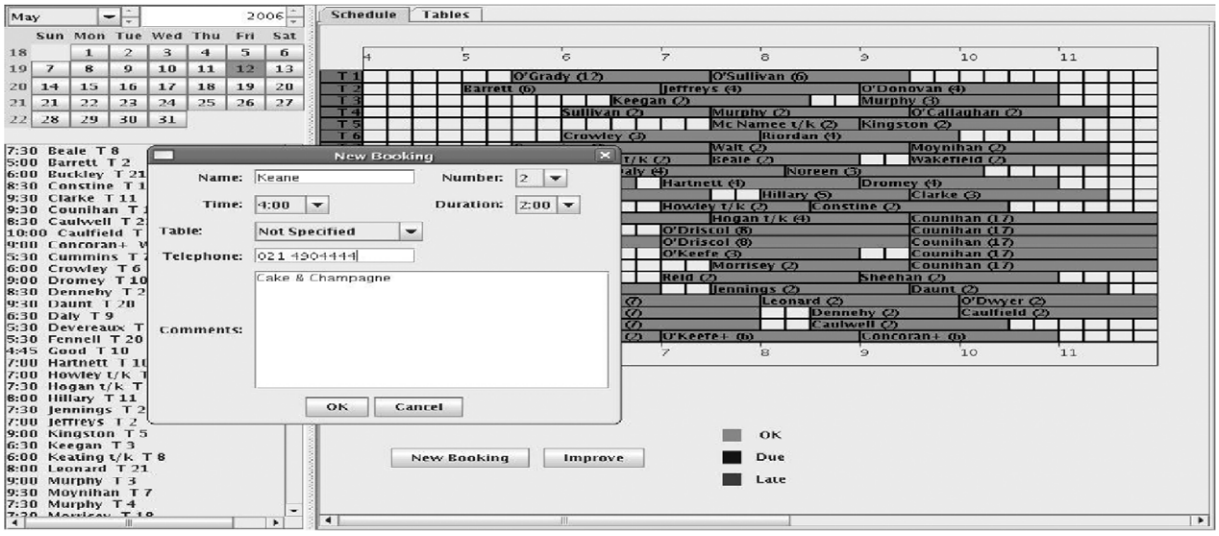

Fig. 5. User interface, displaying a seating plan and a new booking request.

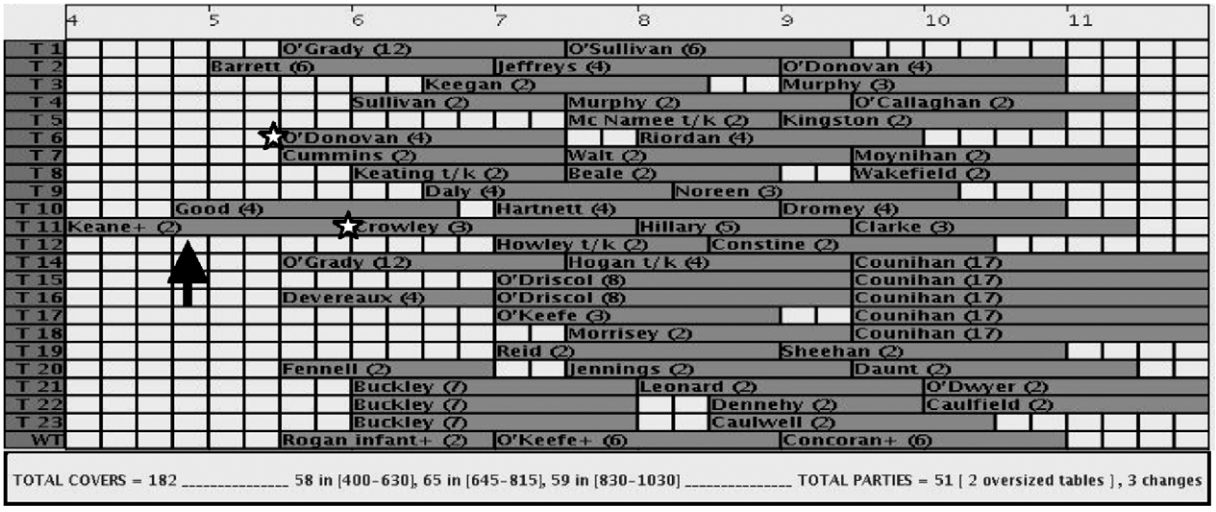

Fig. 6. Seating plan with the new request accommodated into table 4.

Fig. 7, party Keane has been moved to table 11, which is for 5 people. The operation required 3 changes from the previous table allocation.

During booking, availability requests are common – e.g. "when can you seat a party of 4?" The user can process such requests using the same booking form, by selecting "not specified" in the "Time" box. Fig. 8 shows the answer provided by the system for a request for 4 people, for the set of parties illustrated in Fig. 7. The message also groups the available times by the available duration. This is an important information, since the booker may be able to sell the table for one hour at 7 o'clock if the customer is only asking for a quick main course. The procedure that checks the availability is again based on the MH algorithm.

Fig. 9 (top) shows a reallocation of the plan in Fig. 7 that accommodates a new party Meane at 9:00 in table 8. Note that in this case the number of changes necessary to find a new plan is higher, i.e. the system performed a more complex operation. Fig. 9 (bottom) represents a first step in a search for a more flexible allocation. The new plan has been obtained pressing the "Improve" button (Fig. 5). Note that there has been only one change from the previous plan, with party Crowley (3 people at 6:00) moved from table 6 (6-seater) to table 9 (4-seater). The increase in the flexibility estimate is 16 (8 time units × 2 table size saved), which may allow an extra 2-h dinner (8 time units) for 2 people. The run time to obtain the change is 0.16 s.

The user can repeat the improvement process to find more flexible seating plans. Fig. 10 (top) shows the plan obtained after four iterations, and (bottom) the plan obtained unlocking party Keane from table 11 (and after three more iterations).

In both phases, we can observe the effect of our flexibility measure, which by increasing the number of usable start times makes better use of tables, and reduces the unusable

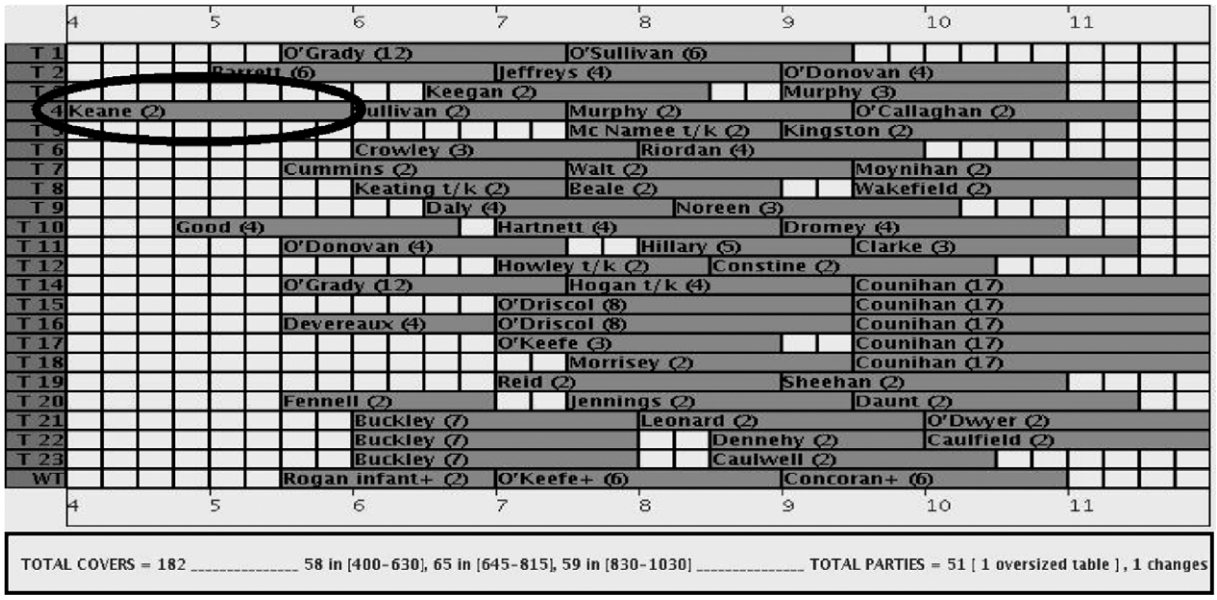

Fig. 7. New seating plan after imposing a preference for party Keane.

Fig. 8. Message showing the availability for 4 people on the sheet of Fig. 7.

zones (empty squares) in between parties. The increase in the flexibility estimate over Fig. 9 (top) is 68 and 96 for Fig. 10 (top) and (bottom). This can be regarded as 3 and 5.5 times the (2 h × 2 people) improvement obtained from the first step of Fig. 9-bottom. The run time from Fig. 9 (bottom) to 10 (top) was 8.1 s, and from 10 (top) to (bottom) was 1.01 s.

Fig. 11 shows an instant during the floor management phase. The current time is represented by the vertical line at 5.30 p.m. Party Keane (table 4) was due to finish, but is going to be late, creating a conflict with the next party Fennell. In this case, the user can edit Keane, extending the duration from 1.30 h to 1.45 h, and ask the system to search for a reallocation that avoids the conflict.

Fig. 12 (top) represents a first reallocation, while on the bottom we see a seating plan after four improvement iterations. We can again observe the benefit of the improvement, with fewer unusable zones, and more possibilities to seat extra parties. The four iterations have improved the flexibility estimate by steps of 4, 5, 4, and 26, for a total of 39, or ~2.5 (2 h × 2 people) dinners. The number of changes from the initial allocation was 2, 1, 3, and 36; the last iteration gave a large improvement but required a large change in the seating plan.

By default, the *timeout* for each improvement step is set to 10 s, partitioned in 7 s for search with limited discrepancy and 3 s for unlimited (or global) search. These limits are configurable by the user.

The research prototype software discussed above is currently being evaluated in the restaurant. The main aim of the evaluation is to determine whether constraint-based methods could support a practical restaurant management tool. Specifically, the evaluation will check that the software:

(i) models the restaurant adequately;
(ii) provides acceptable seating plans in reasonable time;
(iii) can join and separate tables correctly;
(iv) proposes flexible seating plans in reasonable time;
(v) reports quickly whether or not a booking request can be accepted, and recommends sensible alternative times for a booking;
(vi) provides useful advice when a seating plan has to be reconfigured.

If the evaluation is positive (and first indications are promising), then we will investigate commercial development.

8. Conclusions and future work

In this paper we presented a constraint-based solution for enhancing restaurant table management. We introduced the table management problem, describing the main issues concerning booking and floor management. We

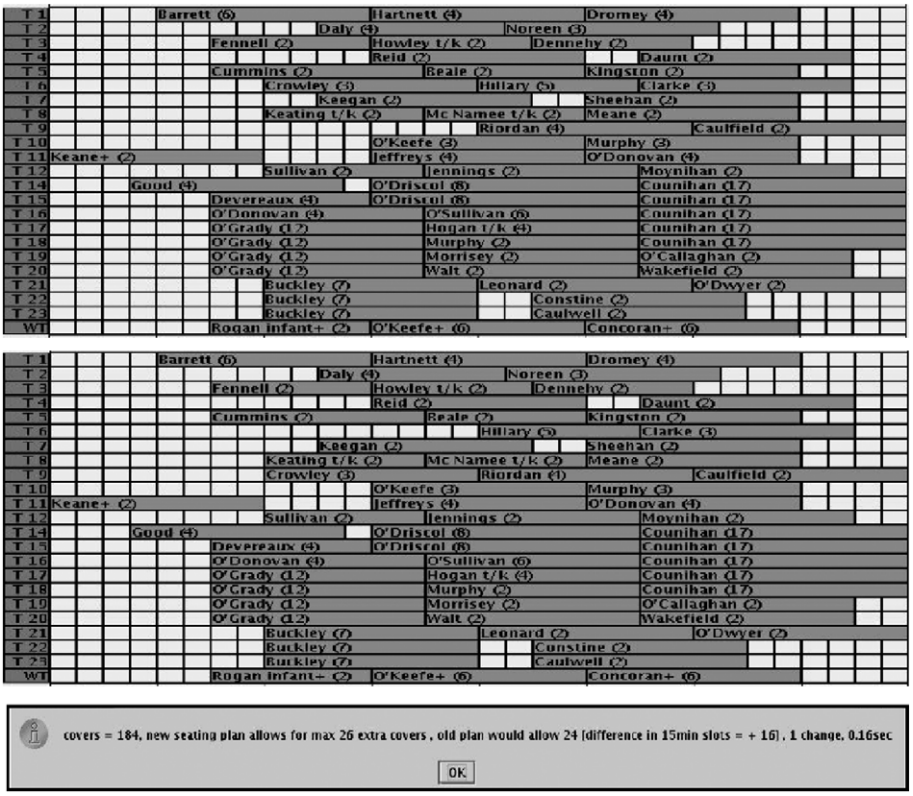

Fig. 9. Adding party Meane (top), first flexibility improvement (bottom).

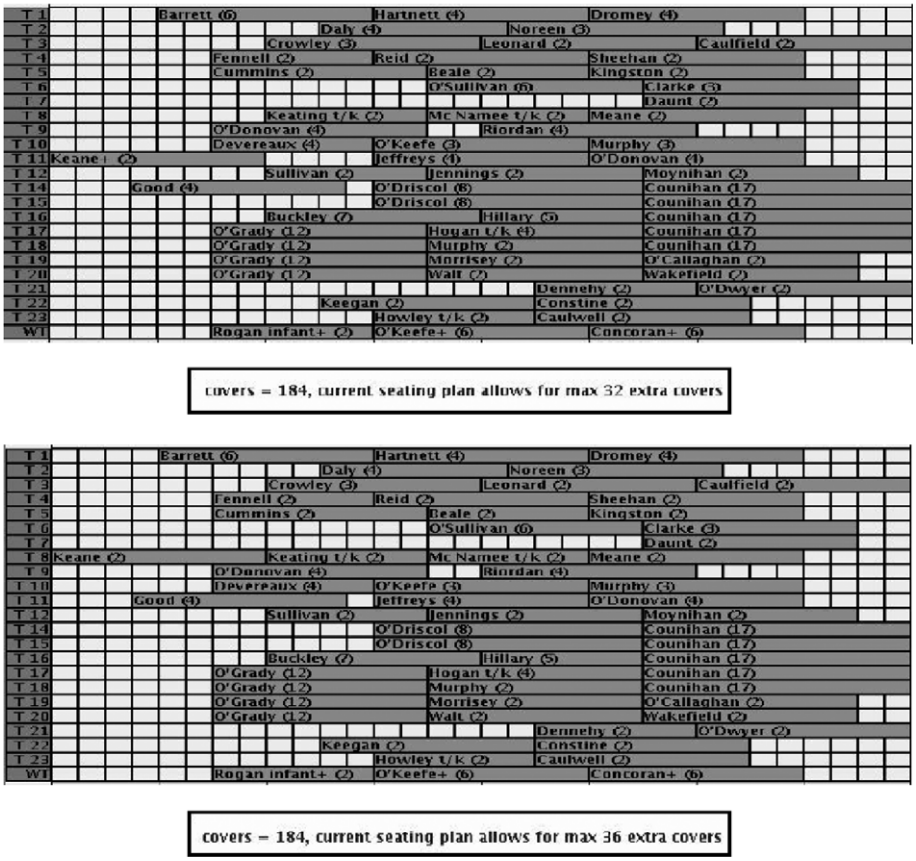

Fig. 10. Improvement after several steps, with Keane fixed (top), unfixed (bottom).

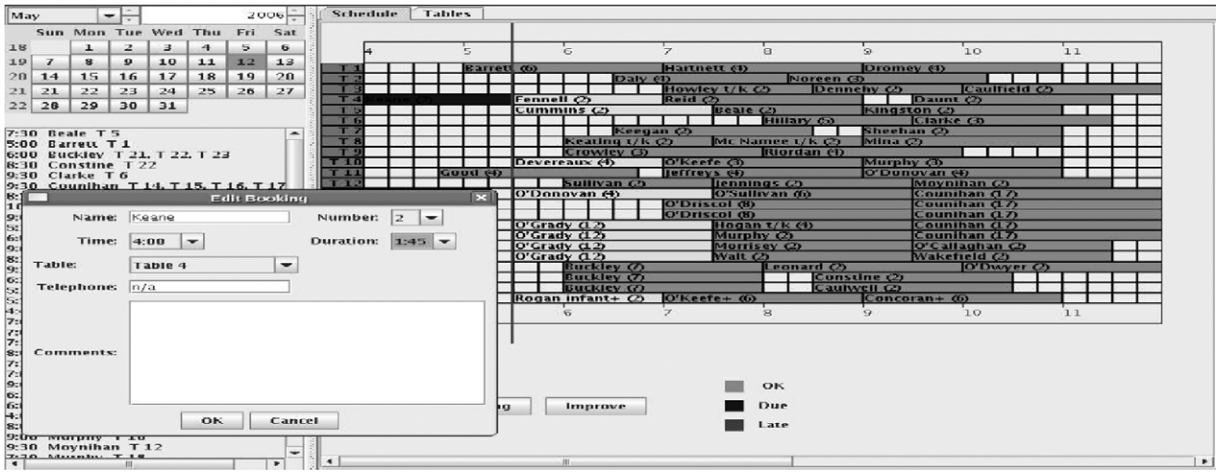

Fig. 11. An instant during floor management, with a late finish (Keane, T4).

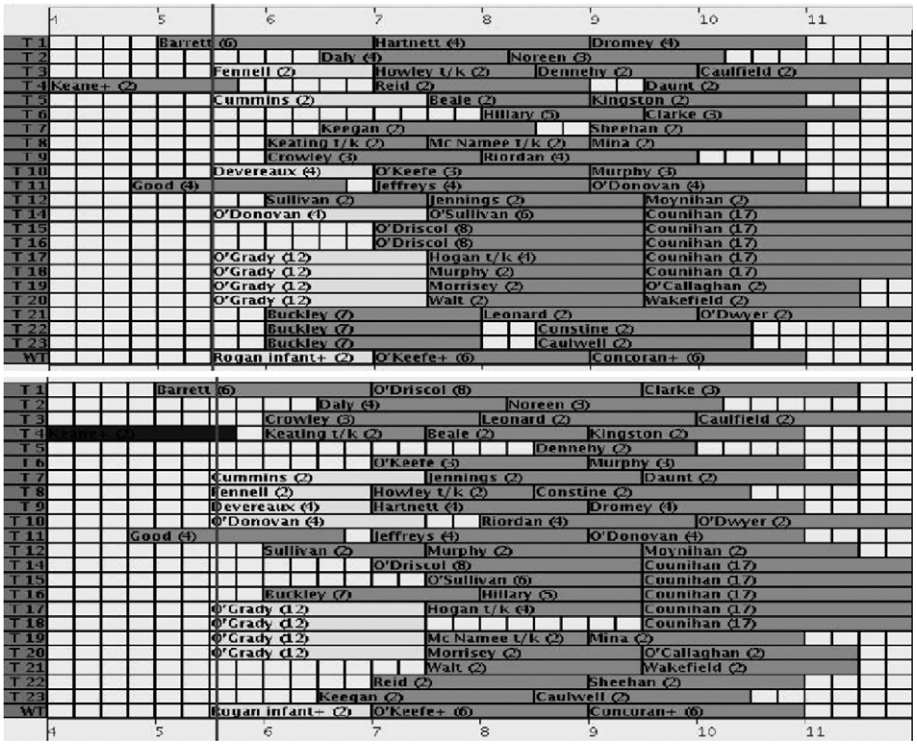

Fig. 12. Reallocation after a late finish (top), improvement after four iterations (bottom).

presented a basic constraint model, which can be used to solve the underlying static problem. We then described two enhancements, which (i) optimise a flexibility measure, and (ii) search for similar plans after a change occurs. The flexibility measure is based on a weighted count of the possible start times for new bookings, and is intended to allow more efficient use of resources. The search for similar solutions minimises the number of changes to the seating plan, and is intended to simplify floor management. We have described the integrated system, which allows a user to control table allocation, while receiving advice from the underlying models. The system has been implemented, and is currently undergoing trials in Eco restaurant.

Future work will focus on improving the flexibility measure, to take into account the expected distribution of demand. Monday evenings, for example, show a noticeably different pattern of dining from Friday evenings, and thus the system should tailor its advice accordingly. Our first approach will be to include weights in the flexibility measure, increasing the importance of availability at specific times. Should the current evaluation trial prove positive, we expect to begin a development phase. This will include

redeveloping the constraint models to ensure they are suitable for the operating environment, and redeveloping the user-interface, based on the feedback from the evaluation.

Acknowledgements

This work is funded by Enterprise Ireland under grant number SC/2003/0081. We are grateful for the problem description, data and advice given by the Eco restaurant in Douglas, Cork. The user interface was developed by James Lupton, and supported by the Science Foundation Ireland Overhead Investment Plan, 2005–2006. Finally, we are grateful for the external liaison assistance of James Little at Cork Constraint Computation Centre.

References

[1] <www.eco.ie/>.
[2] R. Dechter, Constraint Processing, Morgan Kaufman, 2003.
[3] F. Rossi, P. Van Beek, T. Walsh, Eds., Handbook of Constraint Programming, Elsevier, 2006.
[4] C.P. Gomes, D.B. Shmoys, Approximations and randomization to boost CSP techniques, Annals of Operations Research 130 (2004) 117–141.
[5] C.P. Gomes, B. Selman, Algorithm portfolios, Artificial Intelligence 126 (1–2) (2001) 43–62.
[6] A. Vidotto, K.N. Brown, J.C. Beck, Robust constraint solving using multiple heuristics, in: Proc. of the Sixteenth Irish Conference on Artificial Intelligence & Cognitive Science (AICS'05), 2005, 203–212.
[7] J.C. Régin, A filtering algorithm for constraints of difference in CSPs, in: Proc. AAAI-94, 1994, 362–367.
[8] G. Verfaillie, T. Schiex, Solution Reuse in Dynamic Constraint Satisfaction Problems, in: Proc. AAAI-94, 1994, 307–312.
[9] D.W. Fowler, K.N. Brown, Branching constraint satisfaction problems and Markov Decision Problems compared, Annals of Operations Research 118 (1–4) (2003) 85–100.
[10] R. Bent, P. Van Hentenryck, Regrets only! Online stochastic optimization under time constraints, in: Proc. AAAI-04 (2004).
[11] Y. Ran, N. Roos, J. Van Den Herik, Approaches to find a near-minimal change solution for dynamic CSPs, CPAIOR'02, in: Proc. of the 4th International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems, 2002, 373–387.
[12] A. Petcu, B. Faltings, Optimal solution stability in continuous-time optimization, DCR-05, in: Proc. of the 6th International Workshop on Distributed Constraint Reasoning, 2005, 207–221.
[13] A.J. Davenport, C. Gefflot, J.C. Beck, Slack-based techniques for robust schedules, in: Proc. of the Sixth European Conference on Planning (ECP-01), 2001.
[14] E. Hebrard, B. Hnich, T. Walsh, Robust solutions for constraint satisfaction and optimization, in: Proc. of the Sixteenth European Conference on Artificial Intelligence ECAI-04, 2004.
[15] E.M. Arkin, E.B. Silverberg, Scheduling jobs with fixed start and end times, Discrete Applied Mathematics 18 (1987) 1–8.
[16] R.E. Korf, Depth-first iterative deepening: an optimal admissible tree search, Artificial Intelligence 27 (1985) 97–109.