Managing Restaurant Tables Using Constraint Programming

Alfio Vidotto

A Thesis Submitted to the National University of Ireland

in Fulfillment of the Requirements for the Degree of

Doctor of Philosophy.

November, 2007

Research Supervisor: Dr. Kenneth N. Brown and Dr. J. Christopher Beck.
Head of Department: Prof. Gregory Provan.

Department of Computer Science,
National University of Ireland, Cork.

Contents

List of Figures

List of Tables

Abstract

Restaurant table management can have significant impact on both profitability and customer experience. The core of the issue is a complex dynamic combinatorial problem - restaurants must take reservations, and manage unexpected events in real time, making good use of resources, and providing good service to customers. Although the application of artificial intelligence to real world problems has shown a considerable and increasing success in recent years, the problem of restaurant table management has been left largely unexplored.

In this dissertation we develop a solution based on constraint programming to support, enhance, and automate the uncertain and highly dynamic restaurant table management problem. Specifically, our solution allows inexperienced users to take bookings and seat diners, and to automatically reconfigure seating plans; it uses resources, maximizing table usage and final turnover; it enables reaction to unplanned events, minimizing the propagation of delays over future diners; it maintains feasible, flexible, and stable seating plans, simplifying both booking and floor management, in processing table requests, and in tracking and controlling table allocations; it supports the booker with data on time availability, suggesting flexible booking times; and it supports the use of future knowledge, to provide flexible seating plans for expected booking patterns.

Restaurant table management is a new application problem for constraint programming. We represent the problem as multiple machine scheduling with fixed start and end times and reconfigurable machines. We model the problem as constraint satisfaction and develop a search algorithm based on multiple heuristics and time slicing. We extend the model to generate flexible seating plans, and implement a limited discrepancy algorithm to maintain stability when changes occur. The constraint based model presented in this dissertation represents a successful case of research applied to a real world dynamic problem, integrating reaction efficiency, optimization, stability, and robustness. The research underneath this thesis was carried out using information from a real restaurant, was tested using computer simulations, and was finally validated with trials at the restaurant.

Declaration

This thesis is submitted to University College Cork, in accordance with the requirements for the degree of Doctor of Philosophy in the Faculty of Science. The research and thesis presented in this dissertation are entirely my own work. Some parts of the work have been published in the following articles.

1. Vidotto, A., Brown, K. N. and Beck, J. C., Managing Restaurant Tables using Constraints, *Knowledge Based Systems*, March 2007, Vol. 20, Issue 2, Pages 160-169.

2. Vidotto, A., Brown, K. N. and Beck, J. C., Managing Restaurant Tables using Constraints, *Applications and Innovations in Intelligent Systems XIV, Proceedings of the 26th SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence (AI-2006), Applications Stream*, Springer (1 84628 665 4), 2006, Pages 3-16.
   Awarded Rob Milne Memorial Prize for best refereed application paper.

3. Vidotto, A., Brown, K. N. and Beck, J. C., Robust constraint solving using multiple heuristics, *Proceedings of the 16th Irish Conference on Artificial Intelligence and Cognitive Science (AICS-2005)*, Portstewart, Northern Ireland, 2005, Pages 203-212.

4. Vidotto, A., Brown, K. N. and Beck, J. C., Robust constraint solving using multiple heuristics, *Proceedings of the 11th International Conference of Principles and Practice of Constraint Programming (CP-2005), Doctoral Paper*, Sitges, Spain, 2005, Page 871.

Acknowledgements

I would like to thank my supervisors, Dr. Ken Brown and Dr. Chris Beck, for helping me during my PhD. This work was funded by Enterprise Ireland under grant number SC/2003/0081. I am grateful for the problem description, data and advice given by the Eco restaurant in Douglas, Cork. The user interface of the prototype software used during a trial period in Eco was developed by James Lupton, and supported by the Science Foundation Ireland Overhead Investment Plan, 2005 - 2006. Finally, I am grateful for the external liaison assistance of James Little at Cork Constraint Computation Centre.

Chapter 1

Introduction

1.1 Issues in restaurant table management

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 behavior 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 favorite 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 reassign all diners who have not yet been seated to avoid delays? When a party does not appear, can we reassign all other diners to gain an extra seating? In Computer Science terms, table management is an online constrained combinatorial optimization problem - restaurants must manage reservations, and manage unexpected events in real-time, making good use of resources, and providing good service to customers.

On current systems, booking and floor allocations are done manually, and require experienced personnel. The first computerized table management programs have recently started to appear in some busy restaurants, but most of them provide no advice to the user, and do not support dynamic table reallocation. Restaurant table management could be improved if we could develop software that can be used by staff with less expertise and knowledge, and that can help the automation and optimization of the (dynamic) allocation process.

In this dissertation we propose and defend the following thesis:

Constraint programming can be used as a tool to support, enhance, and automate uncertain and highly dynamic restaurant table management.

Specifically, constraint programming can be used:

- *to model table management, and careful modelling can improve efficiency;*
- *to model and solve the underlying static decision problem;*
- *to model table (re)configurations and seating plan flexibility;*
- *for complex or stable seating plan reallocations;*
- *to model knowledge on future demand to build more flexible seating plans;*
- *to exploit diners' start time flexibility to preserve seating plan flexibility;*
- *to improve robustness in managing uncertainty (on demand and changes).*

1.2 Contributions

The research developed in this dissertation provides a practical solution to restaurant table management, supporting flexibility, stability, and robustness. The solution improves current systems in that: it allows inexperienced users to take bookings and seat diners, and to automatically reconfigure seating plans; it allows a more efficient use of resources, maximizing table usage and final turnover, with fewer customers turned away; it supports and improves the reaction to unplanned events, minimizing the propagation of delays over future diners; it maintains feasible, flexible, and stable seating plans, simplifying both booking and floor management, in processing table requests, and in tracking and controlling table allocations; it supports the booker with data on time availability, suggesting more flexible booking times; it is less sensitive to the uncertainty on table demand and on changes; and it supports the use of future knowledge, to provide seating plans adequately flexible with respect to the expected booking patterns.

The research developed in this dissertation is valuable also for the constraint programming community, and for computer scientists in general. Restaurant table management is a new application problem for constraint programming. Like scheduling for table management, many real world problems are dynamic and uncertain. The interest on such problems has been growing considerably in recent years, with new solving techniques being regularly introduced. Typical goals concerning dynamic problem solving are [24]: (i) to provide quick reactions to changes; (ii) to compute the best plan by reasoning on the possible future developments of the problem; (iii) to maintain stable solutions; (iv) to provide solutions that are robust in accommodating changes at little cost. Although real problems often involve multiple goals, current solving techniques tend to focus on one single goal at a time. Further, the integration of research techniques into industrial applications is still in its early phase. The constraint based model presented in this thesis represents a successful case of research applied to an industrial problem, integrating reaction efficiency, optimization, stability, and robustness.

1.3 Overview of the dissertation

The structure of the dissertation is as outlined below.

Chapter 2 Restaurant table management - We present full details of the table management problem. Specifically, we characterize a two phase problem of booking and floor management. We describe common features of the restaurant environment, such as table capacity and combinability, and we introduce general business rules used to control the table allocation process. We discuss the sources of uncertainty, on table demand, customer behavior, and restaurant performance. We introduce the main measures to evaluate the quality of solutions. We finally describe one particular restaurant, and report an example of booking and floor management based on real data.

Chapter 3 Background - We give an introduction to constraint programming, providing an overview of the main literature, with particular interest on extensions for modelling changes and uncertainty in dynamic problems. We then give a description of scheduling, focusing on the subclass of scheduling with fixed start and end times, which we have used to represent restaurant table management. We finally review some research on the more general topic of restaurant revenue management.

Chapter 4 Modelling and solving the static decision problem - We tackle the underlying static decision problem. We model the problem as a subclass of scheduling with fixed start and end times, and characterize it as NP-complete. We present a basic constraint representation, and then extend it with specialized constraints and search algorithms, designed to produce a more efficient and robust search. We conclude showing the benefit achieved through careful modelling, comparing the final advanced solution to the original basic solution.

Chapter 5 Modelling table configurations and seating plan flexibility - We extend the constraint model to represent table configurations, and to allow multiple joining and separation of tables over the same dining session. Then we extend the model to represent seating plan flexibility, and to select seating plans maximizing flexibility - i.e. advising which tables have to be allocated or joined, and when, in order to get a seating plan more flexible for accommodating future table demand. We design three objective functions based on different flexibility measures: the first maximizes usable start times, the second minimizes the time between meals (or dead zones), and the third maximizes the potential number of seatings. Flexibility is then extended to weight the expectation of future table demand over time, with weights generated from booking patterns retrieved from past booking sheets. We design new specialized constraints to speed up the search process with more constraint propagation. We develop search algorithms for anytime solutions, aiming to achieve reasonable flexibility improvements in practical time. We conclude with a test for the selection of the algorithm with the best anytime profile. Again, through careful modelling we achieve an efficient solution.

Chapter 6 Solving the dynamic problem - We tackle the dynamic problem, considering it as a sequence of static problems linked by changes. We present a search algorithm for solution stability involving limited discrepancy search. We test our models over simulated booking and floor management sessions. We compare against simulations of traditional allocation systems. We show that our model is capable of complex reallocations, which allows more flexibility to accommodate future table requests (bookings or walk-ins), and more robustness to accommodate changes (e.g. delays) without delaying other parties. We also show that the solutions we provide are indeed more flexible for the accommodation of future table requests, improving the final number of people the restaurant can accommodate. We show the potential benefit of using our optimization model to exploit customer's start time flexibility, i.e. when some customers are flexible over the time to have their dinner. We evaluate the three flexibility measures designed in Chapter 5, showing they are all accurate in representing the real flexibility. We evaluate flexibility weighted by the expected distribution of table demand (or booking pattern), noticing a further improvement when some level of customer's start time flexibility is also included.

Chapter 7 Restaurant trials - We demonstrate how the interface of the implemented software application provides access to the models and algorithms described in the previous chapters, presents 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, optimizing for flexibility, or maintaining stability. We conclude with a discussion about the evaluation of the software in a real restaurant (Eco [32]).

Chapter 8 Future work - We examine possible extensions of the research. This includes optimizing the multiple heuristic approach developed in Chapter

4 for improving search efficiency and robustness, involving restaurant table mix optimization, designing better flexibility measures based on future table demand, and investigating other approaches to improve robustness or to support stability in managing uncertain events.

Chapter 9 Conclusions - We summarize the work done and the contributions achieved. We conclude with final comments on the thesis defended.

Appendix A - We report the questionnaire compiled by the general manager of the restaurant Eco at the end of the trial period.

Chapter 2

Restaurant table management

2.1 Introduction

In this chapter we present the restaurant table management problem in detail, giving more evidence about its complexity, significant uncertainty, and high dynamism, and describing the traditional solutions. We first characterize a two phase problem of booking and floor management. Then, we present the main ingredients that make it a complex problem, which involves dealing with physical constraints and business rules, on table capacity and combinability, and dealing with the sources of uncertainty, on table demand, customer behavior, and restaurant efficiency. We represent table management as a sequential decision problem, and discuss some measures for evaluating the quality of final solutions, considering the perspective of both the restaurant and the customer. We conclude the chapter by describing a real restaurant, Eco, and an example of booking and floor management, discussing some data and figures taken from a real dinner session.

2.2 The problem

Table management, 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 maximizes the chances of being able to seat the desired number of people. 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. In either phase, decisions have to be taken in real time. As a general guide, we can expect a manager to take an average of 10 to 20 seconds to decide on where to allocate a new booking, or on how to reallocate a seating plan after a change.

2.2.1 Table capacity and combinability

Typically, restaurants have sets of tables of different capacities. Dinners must take place on tables of suitable capacity, e.g. a party of four can only be accommodated into a table of capacity at least four. Some of the table capacities may depend on the state of adjacent tables: for example, two adjacent tables may not be fully occupied at the same time if there is not enough space to fit all the chairs. Some tables may be combinable with others, e.g. in case there is a large party, or if there are too many parties of size 6 to seat on 6-seater tables. There may be different possible layouts (or restaurant configurations) a restaurant can assume, depending on how tables are combined. The capacity of a combined group of tables may not be equal to the capacity of the single tables, e.g. joining two tables for four may not be able to accommodate a group of 8 people. Consequently, the capacity of the restaurant may change according to the layout being used.

The number of configurable layouts can be regarded as a degree of flexibility of the restaurant. More possible layouts means more possibilities to accommodate parties. For example, if *NL* is the number of possible layouts, we can see this as though the restaurant manager can decide how to allocate his customers by picking the most suitable restaurant from *NL* different possibilities. In terms of reasoning (for the manager) or computation (for a computerized allocation system), this number is also a measure of complexity. For a given set of bookings each layout may allow a number of possible ways to allocate the set. Then, if we have to work out the best allocation, we expect it will be approximatively *NL* times harder than considering a single restaurant with no table configuration. Of course, this is the complexity relative to the worst and ideal case, i.e. when we want to find the optimal solution and we assume we know in advance the number, nature, and behavior of all present and future bookings. In the reality, the problem has a high level of uncertainty, e.g. we can only estimate how long a dinner is going to last or which booking requests will arrive next, and the manager uses heuristic approximations to define and allocate the dinner slots.

A restaurant with combinable tables allows multiple joining and separation of tables during the same evening session. For example, a group of combinable tables T_1, T_2, and T_3 may be able to accommodate three parties of two, followed by a party of 5, followed by two parties, one of 3 and one of 2. This means that, if we take a number of pictures on different instants over the final schedule of a standard evening we can end up with a collection of several different layouts. The complexity coming from considering dynamic layouts is therefore higher. For instance, we have to reason not only about the possible restaurant configurations, but also about the sequence of restaurant configurations to operate over time. Again, in the reality table management is realized by using cheaper heuristics. Further, for the floor phase, restaurants may aim to maintain a stable seating plan, i.e. to limit the number of reconfigurations. In fact, if the configuration of tables gets frequently modified the restaurant may become too chaotic, and this can annoy both the staff and especially the customers.

Extending physical constraints with business rules

Table, configuration, and layout capacities represent physical constraints and therefore cannot be violated. Examples of such constraints are: (i) a table for two cannot accommodate four; (ii) the restaurant cannot serve more than two parties of 10 people at the same time because there are only two suitable tables (or groups of combinable tables) which can serve 10 people; and (iii) the number of people eating at the same time cannot exceed the restaurant capacity, considering the highest capacity layout. However, by simply satisfying these physical constraints table management may lead to very poor allocations and profit. For example, reserving many tables for six for parties of two people is feasible but poor (unless it is Valentine's Day). Similarly, large unusable time slots between two consecutive dinners on the same table must be avoided. For example, a booking for a dinner expected to last until 8:30 p.m. should not be allocated to a table already reserved for a dinner starting at 9:15 p.m. as the 45 minutes in between the two dinner slots would not be sufficient for accommodating any extra dinner.

In order to guarantee an acceptable level of turnover, restaurants must aim to maximize the use of their resources (tables) over time. They do this by applying business rules, e.g. limiting the number of twos in oversized tables, or minimizing the time between meals.

2.2.2 Sources of uncertainty

Table management is an online and dynamic problem where partial solutions have to be generated over time and before the complete problem is known. Specifically, the restaurant must manage reservations as they arrive, and manage unexpected events in real-time. There is uncertainty in how the problem develops over time. Table 2.1 reports the main sources of uncertainty, concerning customer behavior and restaurant performance.

Table 2.1: Main sources of uncertainty.

Source	Description
Customer behavior	Table demand (future requests by number, size, and time) Actual arrival time and dinner length Cancellations and no-shows (cancellations without notice) Walk-ins (customers arriving without a booking) Unexpected bookings Changes in booking time and/or size
Restaurant performance	Kitchen and staff efficiency to provide food on time

Table demand

The lack of knowledge about future table demand, i.e. about the distribution of future requests by number, size, and booking time, makes it difficult for the restaurant to build up a seating plan that maximizes the table usage. For example, should a party of two be offered the last four-seater table? The answer depends on the expectation we have on the arrival of parties of size four, but we can never be sure that any party of four will actually arrive.

Arrival and dinner length

Customers rarely arrive at the precise booking time, and the exact length of their stay is also unpredictable. Late arrivals or dinners lasting longer than expected may easily cause delays for future dinners. The restaurant has to preallocate a dinner slot to each customer, but how large should each slot be in order to make sure dinner durations are neither over nor under estimated - i.e. tables are being well utilized, and consecutive dinners allocated on same tables are not going to overlap and produce delays? This is really a gamble between the chance to improve the table usage and the risk of increasing the waiting time of parties that cannot be seated on time. Restaurants use estimates of the expected duration of a meal based on the characteristics of the booking (including time, day of the week, and party size). For example, a dinner at 9:00 p.m. is expected to last longer than one at 4:00 p.m., people typically stay longer on Fridays and Saturdays rather than on Mondays, and the bigger the party the longer (usually) the stay.

Cancellations and no-shows

Every night, in many restaurants, some parties cancel or simply do not turn up. These changes can potentially degrade the profit of an evening session, if the freed dinner slots remain unsold. As a contingency, some restaurants try to maintain a list of reserves, i.e. customers whose requests have been initially rejected but that are willing to be contacted in case a table should become available. Another way to prevent tables from remaining unsold is represented by overbooking, but this is quite risky, and can be undertaken only by experienced staff.

Walk-ins

Even in those nights when a restaurant is initially fully booked, there may be some cancellations, some parties may arrive and get seated earlier, others may leave before the expected time, and thus some tables may become free. These tables can be sold to walk-ins, i.e. customers entering the restaurant with no reservation and asking for immediate availability. There can usually be many of these parties, and their arrival is again unpredictable.

Unexpected bookings

Sometimes it happens that a party arrives believing a booking has been made, but the name does not appear in the booking sheet. Whether the booking has been erroneously placed on the wrong day, or has not been made at all, when the party turns up the manager may have to accept the blame for the mistake and seat the party. The accommodation of the unexpected party may not be possible without delaying future parties - especially if this happens at 8 o'clock on a Saturday night, when a restaurant can be packed and there can be already people with reservation waiting at the entrance.

Booking time and/or size

The booking time or the size of a reservation may change from the initial booking request. Note that a change in booking time can be infeasible if the restaurant is fully booked for the new time, as can be an increase in size if there are no larger tables available. If an infeasible request happens before the dinner session starts, e.g. the customer phones up giving notice of the new booking details, the restaurant can reject it and therefore the current seating plan can be maintained. Disruptions and delays may become necessary instead when such changes happen in dinner time, e.g. if there is a party of 8 with a reservation for 5 at the door, but all the tables that can accommodate 8 have already been reserved.

Kitchen and staff efficiency

Kitchen and staff have a limited and uncertain power of service. Roughly, assuming there is no shortage of staff, a restaurant has the potential to serve a certain (limited) number of parties at the same time - this also depends on the size of the parties, and on the amount and the type of food that has been ordered. Thus, if for example a restaurant can simultaneously serve at most 8 meals, the tendency would be to limit the number of reservations for the same booking time to 8. However, if it happens that one chef calls-in sick, or that customers order a lot of food, the kitchen may no longer be able to provide food on time. In this case, customers have to wait more for the food to be ready, their dinner duration gets stretched, and therefore some following dinners may get delayed.

2.2.3 The sequential decision problem

Restaurant table management is a sequential decision problem: uncertain events occur in sequence, and for each event an action has to be taken (Figure 2.1).

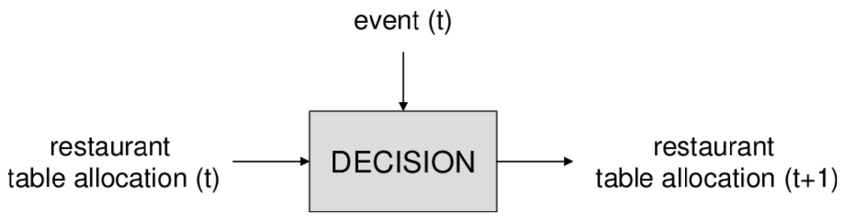

event (t)
restaurant table allocation (t)
DECISION
restaurant table allocation (t+1)

Figure 2.1: Restaurant table management: the sequential decision problem.

Table 2.2 summarizes the possible events during table management, in booking or in the floor phase, and describes the possible decisions for each event type.

Table 2.2: A description of the main events that can occur in restaurant table management. For each event, we report at which stage the event can occur (booking phase = B, floor phase = F), and the corresponding possible decisions.

Possible Event	**Stage**	**Possible Decisions**
new booking request	B F	if there is a suitable table available in the current table allocation then accept; otherwise suggest alternative times / reject
booking change	B F	if the change can be accommodated in the current table allocation then accept; otherwise suggest alternative times / reject
booking cancellation	B F	remove the booking from the current table allocation
walk-in	F	if there is a suitable table available in the current table allocation then accept; otherwise suggest alternative times / reject
no-show	F	if a booking does not arrive after 30 min then remove it from the current table allocation
on-time/late arrival	F	accommodate the party at the earliest available time
early arrival	F	accommodate the party at the earliest available time (even earlier than the booking time if this does not increase delays for other bookings)
dinner shorter than expected	F	no decision is necessary, though the slot of time that has been gained can now be used for future allocations
dinner longer than expected	F	if another dinner was expected to start in the same table (immediately after) then reallocate the second dinner to another table (if possible); otherwise delay it
party turning up with fewer people than the table size that was booked	F	try to reallocate the party to a table of more suitable capacity; otherwise seat the party at the table originally assigned
party turning up with more people than the table size that was booked	F	if the preassigned table is no longer of suitable capacity then accommodate the party to a suitable table as soon as one becomes available
party that arrives believing a booking has been made when none has been recorded	F	accommodate the party at the earliest available time

2.2.4 Quality measures

In restaurant table management, intermediate decisions and partial schedules have to be generated over time aiming to maximize the quality of the final solution, i.e. the results at the end of each evening session. The quality of results is some balance between restaurant income, customer satisfaction, and working conditions. In this section we discuss the main criteria for quality evaluation, considering the restaurant perspective but also the customer perspective. In fact, the manager must make sure customers are also satisfied - the restaurant cannot be happy if the profit of one night was excellent, but no customer is going to come back because the service was disappointing.

The following criteria can be used to give explanation to questions on final solutions, e.g. "how good was last night?", but also to questions on intermediate solutions, e.g. "given the current state at 6 p.m., is the current seating plan expected to favor both customer satisfaction and a good final turnover?"

Quality from the restaurant perspective

From the restaurant perspective, the quality of a schedule (or of a dinner session) can be represented as a function of the number of people the restaurant served over the session. This number is generally called *covers*. The quality grows with the number of covers, but not indefinitely. In fact, more covers means that the work load on the restaurant resources (i.e. tables, kitchen, and staff) increases, however resources have a limited capacity. Further, the restaurant may have gained extra covers (and profit) by packing parties into small tables, or by rushing diners to finish their meal to free the table for the next customer, but this decreases customer satisfaction and therefore cannot be done indefinitely. Figure 2.2 represents a general profile of quality against covers, assuming, for example, a restaurant with an optimal occupancy (or target) of 200 covers.

Restaurants can also estimate the quality of a schedule with respect to the *closing time*. The closing time should not be too early, otherwise it would mean that it has been a very quiet night, and possibly not too late, to avoid paying extra hours to the staff. For example, it may be better closing at 11:45 p.m. having served 170 customers than closing at 00:15 a.m. serving 180.

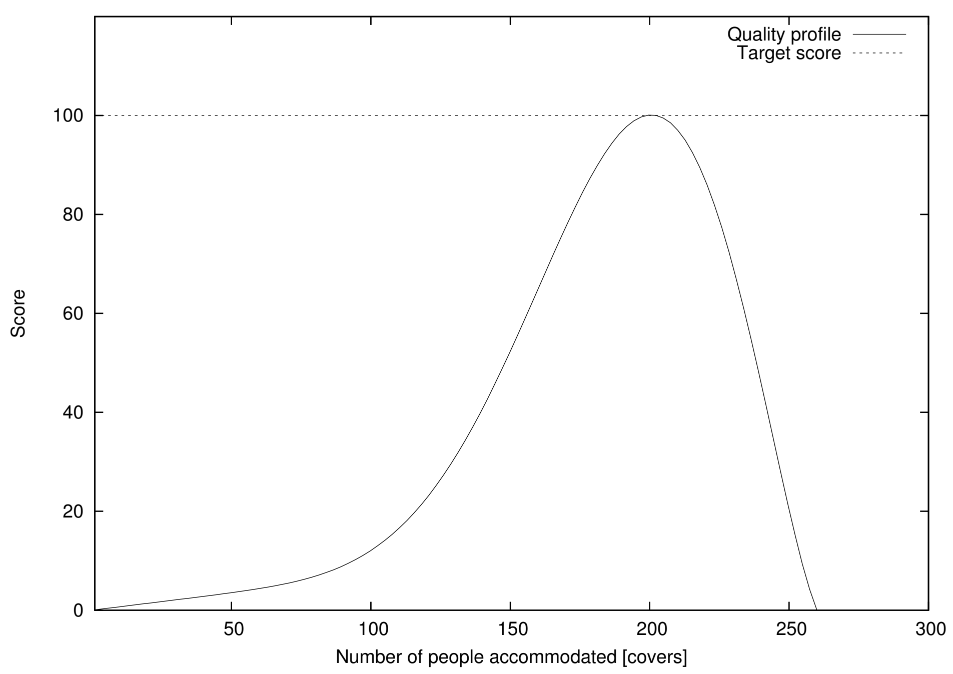


Figure 2.2: General profile of quality against number of covers, assuming an optimal occupancy of 200 covers.

Quality must also be a function of *profit*. Obviously, considering the short term target of one night, the quality must increase proportionally with the profit. However, in some cases a restaurant may prefer to degrade the profit of the night in order to satisfy an important customer. For example, they may reserve extra-spacious tables to VIP parties. Similarly, a restaurant may reserve a table to a regular customer even if the party has not yet confirmed his arrival - with the risk that the table may remain unsold. Of course, VIP and regular customers are very important, and a loss in profit for a night may be worthwhile in order to ensure a higher profit for the long term.

Quality from the customer perspective

A first index of quality for the customer is characterized by the *waiting time*. Restaurants should aim to seat parties as soon as possible after the arrival time, or at least within 15 minutes from the booking time. A delay of half an hour becomes very annoying, and longer delays can be unacceptable - the customer is likely to go away, and perhaps never come back. Figure 2.3 shows a general profile of quality against waiting time, where we give a score from 0 (completely unacceptable) to 100 (completely acceptable) to the possible waiting times.

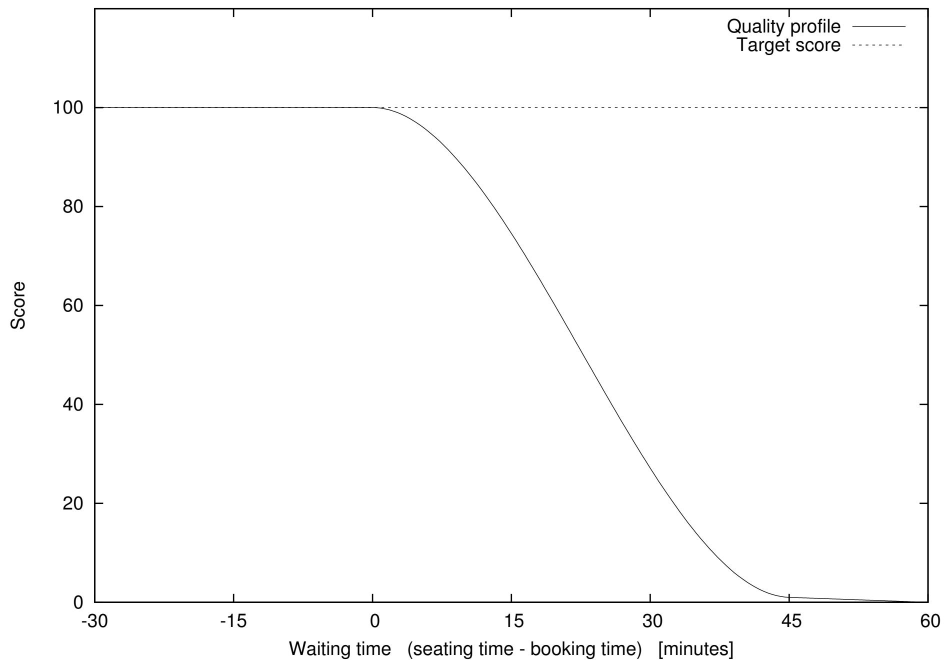


Figure 2.3: General profile of quality against waiting time.

A second index of quality is *service efficiency*, represented by the time a party have to wait for the food (e.g. first course) to be ready and delivered. For example, the restaurant should aim to allow between 5 to 15 minutes to serve the first course, less than that and the customer may feel pressured, more than that and he may get annoyed.

A third index of quality is represented by *table preference*. Customers like to dine in comfortable tables, neither too tight nor too large. For example, parties of four usually prefer to seat into tables of capacity in the range 4 to 6, even though larger tables can also be used. Occasionally, e.g. as a last resort when the restaurant has been overbooked, four people may be accommodated in a table that normally serve 3, but this becomes very uncomfortable. Restaurants may have to satisfy specific preferences, especially if they concern regular or important customers, e.g. reserve a more comfortable table to the bank manager.

2.3 Case study: the Eco restaurant

Eco [32] 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. Figure 2.4 shows the restaurant table map.

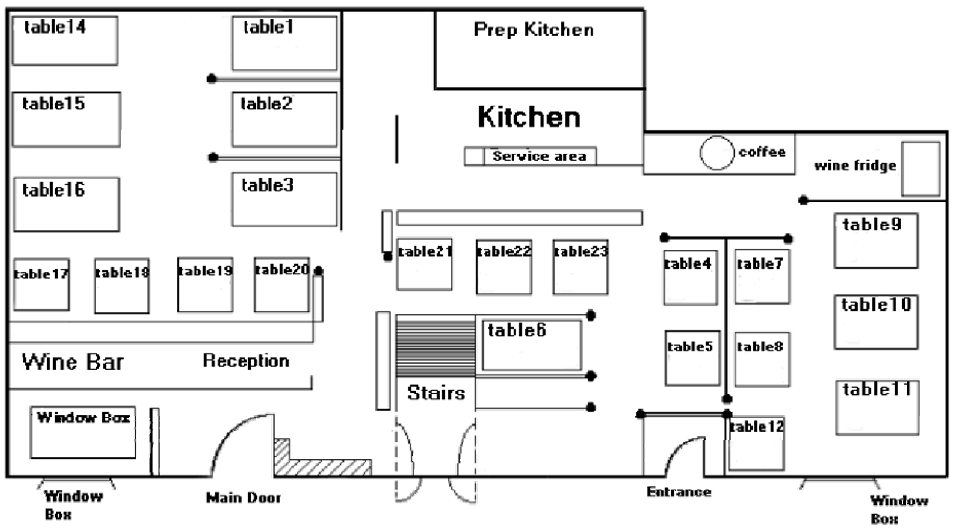


Figure 2.4: Table map of the restaurant Eco.

The restaurant has 23 tables, ranging in size from 2 to 8. 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 to 5 diners. The maximum party size that can be seated at a conjoined table is 30. There are 386 different possible restaurant configurations, and thus the restaurant capacity ranges from 83 to 96. 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 180 and 210 covers (individual diners) each evening - fewer than that, and the tables are not being well utilized; more than that, and the kitchen will be stretched to provide the food on time.

Table 2.3 displays the capacity of each table in the Eco restaurant, and reports those cases where the capacity depends on the state of other tables. Note that, as named in Eco, table WT (window table) is the one located in the window box (see Figure 2.4). Table 2.4 shows the 16 possible table configurations in Eco, along with the range of capacity each one is used for.

Table 2.3: Single capacity of the 23 tables in Eco.

Tables	**Capacity**	**Particular case capacity**
T1	6	max is 4 if T14 seats 5 or 6
T2	6	max is 4 if T15 seats 5 or 6
T3	3	-
T4	2	-
T5	2	-
T6	6	-
T7	2	-
T8	2	-
T9	4	-
T10	4	-
T11	5	-
T12	2	-
T14	6	max is 4 if T1 seats 5 or 6
T15	6	max is 4 if T2 seats 5 or 6
T16	7	-
T17	4	-
T18	2	-
T19	2	-
T20	2	-
T21	2	-
T22	2	-
T23	2	-
WT	8	-

There are different possible layouts (or restaurant configurations) the restaurant can assume depending on which (if any) of the 16 table configurations are used. Note that, in Table 2.4 some tables appear in more than one of the 16 groups, therefore the maximum number of simultaneous table configurations must be less than 16. For instance, there can be layouts with no configuration, i.e. using all the 23 tables as single tables, and with one or more configurations, up to 6 different

Table 2.4: Capacity of the 16 table configurations in Eco.

Possible configurations	Capacity range
T1 + T14	7 to 12
T5 + T6	7 to 9
T14 + T15	8 to 11
T14 + T15 + T16	12 to 16
T14 + T15 + T16 + T17 + T18	17 to 23
T14 + T15 + T16 + T17 + T18 + T19 + T20	24 to 30
T15 + T16	8 to 11
T17 + T18	5 to 7
T17 + T18 + T19	8 to 11
T17 + T18 + T19 + T20	12 to 16
T18 + T19	3 to 4
T18 + T19 + T20	5 to 8
T19 + T20	3 to 4
T21 + T22	3 to 5
T21 + T22 + T23	6 to 10
T22 + T23	3 to 5

from the set of 16. An example of layout with 6 configurations at the same time can be obtained by joining table 1 and 14, table 15 and 16, table 17 and 18, table 19 and 20, table 21 and 22, and table 5 and 6. Table 2.5 displays the number of possible layouts by increasing number of table configurations operating simultaneously. We can see that the restaurant allows for a total of 386 different layouts. Moreover, depending on the layout the overall capacity of the restaurant ranges from 83, when all tables are utilized singularly, to 96, obtained by joining tables 1 and 14, tables 17, 18, 19, and 20, tables 21, 22, and 23, and tables 5 and 6.

In order to guarantee an acceptable level of turnover, the Eco restaurant applies the business rules reported in Table 2.6. For instance (*rule a*), the restaurant should aim not to waste any table for four or more by allocating it with any party of two, especially if bigger parties are likely to arrive. Further (*rule b*), large unusable time slots between two consecutive dinners on the same table must be avoided. 8 o'clock bookings are also regarded as poor (*rule c*). In fact, as the standard duration for a dinner at 8 o'clock is at least two hours, and because very few

Table 2.5: Possible restaurant layouts in Eco.

Table configurations	**Possible layouts**
0 (all tables are single)	1
1	16
2	78
3	144
4	119
5	25
6	3
7+	0

tables are usually sold after 10 p.m., 8 o'clock bookings are likely to prevent tables from serving any more parties later on. These bookings are very rarely accepted, and often a different time is arranged (e.g. 7:30 or 8:30) in order to preserve the potential number of seatings. The restaurant aims to achieve three dinners per table at the end of an evening session (*rule d*). Note that this target becomes easier to achieve if we make sure no table has large idle times and if we do not accept any 8 o'clock bookings. As a final point (*rule e*), the restaurant very rarely sells any table to any party of a single person, especially when the restaurant is or is expected to be busy.

Table 2.6: Main business rules in Eco.

Rules description
a) no or very few parties of two into 4+ seater tables
b) no large unusable time slots between two consecutive dinners
c) none or very few bookings at 8 o'clock
d) three seatings per table
e) no or very few parties of one person

The restaurant deals with special requests and table preferences, and there are priorities depending on who is making a requests. Table 2.7 summarizes the main examples.

Table 2.7: Special requests and table preferences in Eco.

Case	Description
request from VIP	accepted even if there are no tables available (some parties will have to be delayed), and accommodated into comfortable tables (e.g. using a 4-seater even if it is for a couple)
table preference	any party can book a specific table, if this allows to maintain a good seating plan
special tables preference	preferences also concern special tables (e.g. tables 1, 2, and 6, which are booths)
restaurant sides preference	there is an old side (more quiet), and a new side (more noisy)
window table preference	more private, but near the door (therefore can be cold)
buggies or wheel chairs	require tables of extra size
families with kids	may fit into smaller tables

2.3.1 An example of booking and floor management

Both booking and floor management act in accordance to the quality criteria introduced in Section 2.2.3, i.e. reservation and unexpected events are processed aiming to maintain a seating plan that can possibly favor a good final score on all covers, closing time, profit, waiting time, service efficiency, and table preference. In this section we discuss the figures of one example regarding one dinner session at the Eco restaurant. This will give more evidence about the way bookings and uncertain events are handled in the real case. The data concerns one Monday night in early 2006.

Booking sheet and seating plan

Shortly before the opening time (which is scheduled at 4 p.m.), the sheet with the bookings for the night passes from the reservation office to the restaurant reception. Figure 2.5 reproduces the booking sheet prepared by the booker. Each row is a table over time, and there are four groups of columns which identify the seating position of each party on a table. The name of the parties are written into an appropriate seating slot, along with the booking time and the size.

Each party is allocated by the booker into a table (or set of tables) of suitable capacity. For example, party Forde booked a dinner for 10 people for 7 o'clock, and has been allocated into the table configuration composed by table 1 and 14. Further, the restaurant seating plan shows a dynamic layout over the night, e.g. the group of tables 17, 18, and 19 are kept separate for their first seating and are then joined to serve party O'Sullivan, which is expected for 7:30 p.m.

Only three bookings for two people are allocated into tables for four or more. One of these bookings was a special case, i.e. Nicky and Jean requested a booth (table 6) for their wedding anniversary. Dinner slots are allocated so that they do not create any large unusable time slots. For example, party Derry (4:30 p.m.) is allocated in the same table of party Hegarty (6:30 p.m.), and not with Owen (7:30 p.m.), as the expected dinner duration for Derry is 2:00 hours. Finally, there is only one booking at 8 o'clock (Murphy), all tables have still the potential to serve three seatings, and there are no bookings of size one.

From initial seating plan to final allocation

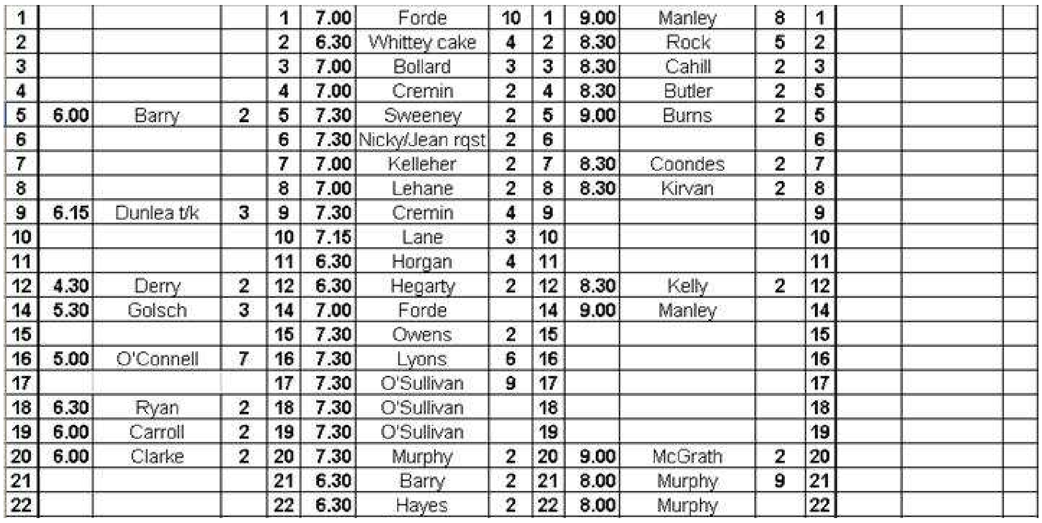
When the evening session starts, the booking sheet represents an initial allocation plan. Figure 2.6 represents the final restaurant allocation, i.e. each party is positioned into the same table it was eventually seated at the restaurant. The parties that maintained the original allocation are highlighted in gray.

Comparing final allocations to the initial plan

Many changes happened between the initial plan and the final allocation. For comparison, Table 2.8 reports some statistics about the two sheets.

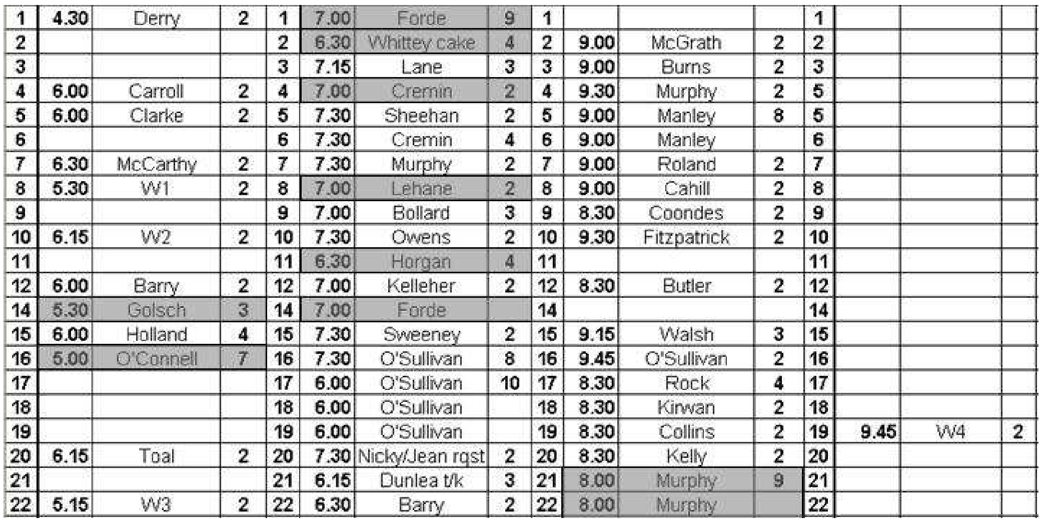
Table 2.8: Dinner session: end-to-end allocation statistic.

	Covers	**Parties**	**Covers/Parties**
Initial plan	126	38	3.32
Cancellations	6 (5%)	3 (8%)	2.00
Final Allocation	159	50	3.18
Walk-ins	42 (26%)	15 (30%)	2.80
Allocations changed from initial plan	79 (63%)	27 (71%)	2.93



1				1	7.00	Forde	10	1	9.00	Manley	8	1			
2				2	6.30	Whittey cake	4	2	8.30	Rock	5	2			
3				3	7.00	Bollard	3	3	8.30	Cahill	2	3			
4				4	7.00	Cremin	2	4	8.30	Butler	2	5			
5	6.00	Barry	2	5	7.30	Sweeney	2	5	9.00	Burns	2	5			
6				6	7.30	Nicky/Jean rqst	2	6				6			
7				7	7.00	Kelleher	2	7	8.30	Coondes	2	7			
8				8	7.00	Lehane	2	8	8.30	Kirvan	2	8			
9	6.15	Dunlea t/k	3	9	7.30	Cremin	4	9				9			
10				10	7.15	Lane	3	10				10			
11				11	6.30	Horgan	4	11				11			
12	4.30	Derry	2	12	6.30	Hegarty	2	12	8.30	Kelly	2	12			
14	5.30	Golsch	3	14	7.00	Forde		14	9.00	Manley		14			
15				15	7.30	Owens	2	15				15			
16	5.00	O'Connell	7	16	7.30	Lyons	6	16				16			
17				17	7.30	O'Sullivan	9	17				17			
18	6.30	Ryan	2	18	7.30	O'Sullivan		18				18			
19	6.00	Carroll	2	19	7.30	O'Sullivan		19				19			
20	6.00	Clarke	2	20	7.30	Murphy	2	20	9.00	McGrath	2	20			
21				21	6.30	Barry	2	21	8.00	Murphy	9	21			
22				22	6.30	Hayes	2	22	8.00	Murphy		22			

Figure 2.5: Booking sheet representing the initial seating plan.



1	4.30	Derry	2	1	7.00	Forde	9	1				1			
2				2	6.30	Whittey cake	4	2	9.00	McGrath	2	2			
3				3	7.15	Lane	3	3	9.00	Burns	2	3			
4	6.00	Carroll	2	4	7.00	Cremin	2	4	9.30	Murphy	2	5			
5	6.00	Clarke	2	5	7.30	Sheehan	2	5	9.00	Manley	8	5			
6				6	7.30	Cremin	4	6	9.00	Manley		6			
7	6.30	McCarthy	2	7	7.30	Murphy	2	7	9.00	Roland	2	7			
8	5.30	W1	2	8	7.00	Lehane	2	8	9.00	Cahill	2	8			
9				9	7.00	Bollard	3	9	8.30	Coondes	2	9			
10	6.15	W2	2	10	7.30	Owens	2	10	9.30	Fitzpatrick	2	10			
11				11	6.30	Horgan	4	11				11			
12	6.00	Barry	2	12	7.00	Kelleher	2	12	8.30	Butler	2	12			
14	5.30	Golsch	3	14	7.00	Forde		14				14			
15	6.00	Holland	4	15	7.30	Sweeney	2	15	9.15	Walsh	3	15			
16	5.00	O'Connell	7	16	7.30	O'Sullivan	8	16	9.45	O'Sullivan	2	16			
17				17	6.00	O'Sullivan	10	17	8.30	Rock	4	17			
18				18	6.00	O'Sullivan		18	8.30	Kirwan	2	18			
19				19	6.00	O'Sullivan		19	8.30	Collins	2	19	9.45	W4	2
20	6.15	Toal	2	20	7.30	Nicky/Jean rqst	2	20	8.30	Kelly	2	20			
21				21	6.15	Dunlea t/k	3	21	8.00	Murphy	9	21			
22	5.15	W3	2	22	6.30	Barry	2	22	8.00	Murphy		22			

Figure 2.6: Booking sheet representing the final restaurant allocation. The allocations that have not changed from the initial seating plan appear highlighted in grey.

The initial booking sheet counted 38 parties, for a total of 126 covers. There were 3 cancellations before or during the night, for a total of 6 people. In the final restaurant allocation there were 50 parties, corresponding to 159 people. During the night 15 walk-in parties were allocated, for a total of 42 people, which corresponds to 26% of the total number of covers. Note that 27 out of 38 parties (i.e. 71%) got seated into a table different from the initial plan. This happened in order to find space for walk-ins, to accommodate delays and other unexpected events, and to maintain appropriate allocations of parties to table. The manager says that the number of changes from the initial plan is particularly higher on Mondays, Tuesdays, and Wednesdays, but it gets much lower on weekends. In fact, at the weekend the restaurant is busier and the initial booking sheet is already almost full, thus there is less flexibility for reallocations but also less necessity to make space for walk-ins. Further, on weekends the restaurant gets under pressure, therefore reducing the changes makes the allocation job easier, helps keep everything under control, and prevents both customers and staff from getting annoyed and stressed by chaotic reallocations and continuous reconfigurations of tables. Finally, note how the mean number of covers per party is slightly decreased at the end of the night. In fact, most of the parties walking in were twos.

2.4 Chapter summary

In this chapter, we have described restaurant table management, viewing it as a two phase problem of booking and floor management. We have introduced the main physical constraints that characterize the restaurant environment, concerning table capacity and combinability, and discussed how restaurants use business rules to control the allocation process, to ensure an acceptable table usage. We then described the sources of uncertainty, concerning table demand, customer behavior, and restaurant efficiency. We discussed the principal measures for evaluating the quality of final solutions, considering the perspective of both the restaurant and the customer. We finally introduced a real restaurant, Eco, and an example of booking and floor management, discussing some data and figures taken from a real dinner session. In the next part of this thesis we are going to develop and test our model of the problem, using the restaurant Eco as case study.

Chapter 3

Background

In this chapter, we start with an introduction to *Constraint Satisfaction Problems* (CSPs), describing the fundamental concepts for representing and solving CSPs, and reviewing the state of the art about *Constraint Programming* techniques. We then introduce the scheduling problem, focusing on the subclass that we will use to represent our restaurant application, i.e. scheduling jobs with fixed start and end times. We conclude with a review of the relevant literature concerning restaurant management.

3.1 Constraint Programming

Constraint Programming (CP) is a powerful AI problem solving technique whose application domain has been growing considerably in the past decade. In particular, CP has been used with success to model many real world combinatorial problems, like resource allocation, scheduling, routing, and configuration. A detailed introduction to CP can be found in [33], while [81] surveys recent research.†

In general, modelling a problem involves two main issues:

$$\begin{array}{ll} (i) & how\ do\ we\ \textbf{represent}\ the\ problem? \\ (ii) & how\ do\ we\ \textbf{solve}\ the\ problem? \end{array}$$

†Unless otherwise noted, the information presented in this section can be found in [33].

A constraint program **represents** the problem as a *Constraint Satisfaction Problem* (CSP), and then **solves** the CSP with a combination of constraint propagation and search (typically backtracking search).

3.1.1 Basic CSP representation

In CP terms, representing a problem means selecting a set of *decision variables*, each with a *domain of values*, and a *set of constraints* over these variables that restricts the values that subsets of variables can take.

A CSP is then a triple $\langle X, D, C\rangle$ defined by:

$$X = \{X_1,\ X_2,\ ...,\ X_n\}\ ,\ \text{set of decision variables;}$$

$$D = \{D_1,\ D_2,\ ...,\ D_n\}\ ,\ \text{corresponding domains of values;}$$

$$C = \{C_1,\ C_2,\ ...,\ C_m\}\ ,\ \text{set of constraints.}$$

Each constraint C_i is defined by a *scope* (S_i), i.e. an ordered subset of variables $\langle X_{i1}, X_{i2}, .., X_{ik}\rangle$, and by a *relation* $R_i \subseteq D(X_{i1}) \times D(X_{i2}) \times .. \times D(X_{ik})$, which defines the allowed tuples of values for the scope. The number $k \in \{1, 2, .., n\}$ is called *arity* of the constraint. In particular, we have: *unary* constraints, involving a single decision variable; *binary* constraints, involving two variables; and *non-binary* constraints, involving $n > 2$ variables. Finally, *binary* CSPs are CSPs where all constraints are binary, while *non-binary* CSPs are CSPs which are not *binary*.

Constraints are naturally stated using equations, logical relations, or other mathematical representations, which are implicit versions of the *scope-relation* form introduced above. For example, given 3 variables, X_1, X_2, and X_3, with initial domains $D_1 = D_2 = D_3 = \{3, 4, 5\}$, possible constraints are:

$$C_{unary} : X_1 \neq 4\ ,$$

$$C_{binary} : X_1 < X_2\ ,$$

$$C_{global} : \text{all-different}(X_1, X_2, X_3)\ .$$

The correspondent scopes and relations are expressed as follow:

$$\begin{aligned}
C_{unary} &: \ S = \langle X_1 \rangle, \ R = \{3, 5\} \ ; \\
C_{binary} &: \ S = \langle X_1, X_2 \rangle, \ R = \{(3, 4), (3, 5), (4, 5)\} \ ; \\
C_{global} &: \ S = \langle X_1, X_2, X_3 \rangle, \\
& \quad R = \{(3, 4, 5), (3, 5, 4), (4, 3, 5), (4, 5, 3), (5, 3, 4), (5, 4, 3)\} \ .
\end{aligned}$$

The *all-different* constraint affects X_1, X_2, and X_3, and it belongs to the class of *global* constraints. Global constraints are defined to act on a collection of variables. In terms of the problem described above, a global constraint is no different from any other constraint. But in terms of constraint functions in a library they are different. Normal constraints have a fixed arity, e.g. "$\leq$" has an arity of 2. A global constraint has no fixed arity - it can be applied to a collection of variables.

An *assignment* to X (or *state*) is an assignment of values to some or all of the variables in X:

$$\{X_1 = v_1, \ X_2 = v_2, \ .. \ , \ X_i = v_i\}, \ v_k \in D_k, \ 1 \leq k \leq i, \ i \leq n \ .$$

An assignment of $i < n$ variables is called a *partial* assignment, while for $i = n$ variables we have a *complete* assignment.

An assignment (partial or complete) *satisfies* a constraint C if the tuple of values in the assignment $\{v_1, v_2, .., v_i\}$ is allowed in the constraint relation R_C. Continuing the example above, a partial assignment $\{X_1 = 5, X_2 = 3\}$ does satisfy C_{unary} and C_{global}, as the relations of the two constraints contain, in the order, $\{5\}$ and $\{(5, 3, 4)\}$. For the same assignment, instead, constraint C_{binary} is not satisfied, as $\{5, 3\}$ does not appear in its relation.

An assignment that satisfies all constraints is called a *consistent* (or *legal*) assignment. A *solution* to a CSP is a complete and consistent assignment, i.e. an assignment to all of the variables,

$$\{X_1 = v_1, \ X_2 = v_2, \ .., \ X_n = v_n\}, \ v_k \in D_k, \ 1 \leq k \leq n \ ,$$

that satisfies all the constraints.

CSP tightness, density, and size

Constraints (and CSPs) are classified in terms of *tightness* (t), depending on the number of tuples they disallow. For example, given two variables X_1 and X_2, both with initial domains $\{1, ..., 10\}$, we can observe how the following four constraints have considerably different (and increasing) tightness:

$$
\begin{aligned}
C_1 &: \neg(X_1 = 2 \vee X_2 = 7) , \\
C_2 &: X_1 \neq X_2 , \\
C_3 &: X_1 < X_2 , \\
C_4 &: (X_1 = 1 \wedge X_2 = 3) \vee (X_1 = 8 \wedge X_2 = 9) .
\end{aligned}
$$

A $CSP(X, D, C)$ is conceptually represented as a *constraint graph* $G(V, E)$, where each variable $X_i \in X$ corresponds to a node $V_i \in V$, and for every set of variables connected by a constraint $C_j \in C$ there is a corresponding hyper-edge $E_j \in E$. The number of hyper-edges connected to each node is called the *degree* of the node (or variable). CSPs are classified also in terms of *density* (d), depending on the percentage of edges compared to the total possible.

The size of a CSP is the size of the *search space*, i.e. $D_1 \times D_2 \times ... \times D_n$, or all possible combinations of values to the set of variables. Consider, for simplicity, a CSP composed of a set of n variables, each with an initial domain of m values. In this case the pair (n, m) is sufficient to define the problem *size* - for the example above, the 2 variables with domain size 10 would generate a search space of size 100. The average difficulty of solving randomly generated CSPs has been related to the quadruple $\langle n, m, d, t \rangle$ (discussed in Section 3.1.4).

3.1.2 Constraint propagation

In constraint propagation, the domains of decision variables are reduced by removing values which cannot appear in any solution. 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 the search tree that has to be explored.

A large part of the success of constraint programming tools is due to efficient domain filtering algorithms for specialized constraints, e.g. the global *all-different* constraint [80]. Propagation on global constraints is particularly effective - a single global constraint connects together many nodes of the constraint graph, so many of the variables get directly affected.

Local consistency

Constraints are propagated using specific algorithms. Depending on the algorithm adopted, propagation makes the problem achieve a certain level of *consistency*. Typically, considering a constraint graph representation of CSPs, only domains concerning nodes within a limited path (e.g. one edge) from each other are maintained consistent. For this reason we talk about *local consistency*. A lot of work has been done to define and compare different types of local consistency [31], for binary and (partially) for non-binary CSPs.

Arc-consistency (AC) [14] concerns binary constraints. A binary constraint $C(X_1, X_2)$ is AC if and only if for every value for X_1 there is a consistent value (or *support*) for X_2 and vice versa. A problem is AC if and only if every constraint is AC. For example, if we consider the domains $D_1 = \{1, .., 6\}$, and $D_2 = \{2, .., 4\}$, then the constraint $C(X_1, X_2)$: $X_1 < X_2$ is not arc consistent. In fact, for X_1 equal to 4, 5, or 6, there is no value for X_2 satisfying the constraint. Constraint propagation would make the problem AC by reducing the domain of X_1 to $D_1 = \{1, .., 3\}$.

(i, j)-*consistency* [39] is a more general type of consistency for binary CSPs. A problem is (i, j)-consistent if and only if any consistent instantiation of i variables can be consistently extended to any other j variables. Note that a problem is AC if and only if it is $(1, 1)$-consistent. Further, a problem is *strongly* (i, j)-*consistent* if and only if it is (k, j)-consistent for any $k = 1..i$.

Path-consistency (PC) [69] is a sub-class of (i, j)-consistency. A problem is PC if and only if it is $(2, 1)$-consistent. Further, a problem is *strongly-path-consistent* (*strongly*-PC) if and only if it is AC and PC. A problem is *path-inverse-consistent* (PIC) if and only if it is $(1, 2)$-consistent.

Generalized-arc-consistency (GAC) [70] generalizes the concept of AC to non-binary CSPs. A constraint is GAC if, for every value for each variable there exists a tuple of values for all the other variables in the scope of the constraint that satisfies the constraint. A problem is GAC if all its constraints are GAC.

In [31] a concept of *consistency tightness* was defined. A-consistency is tighter than B-consistency (written "$A \geq B$") if and only if any A-consistent problem is also B-consistent. Also, $A > B \iff (A \geq B) \wedge \neg(B \geq A)$. The authors showed that some types of consistency are not comparable, but they proved the following relations: *strongly*-PC $>$ PIC $>$ GAC.

One of the most popular local consistency algorithms is AC-3 [65] [66], which transforms a CSP into its arc-consistent equivalent in $O(ed^3)$, with e number of constraints, and d maximum domain size. A more recent version is AC-2001 [17], which extends AC-3 with auxiliary data structures used to record consistency checks, and achieves a time complexity bound of $O(ed^2)$. Many other algorithms have been introduced over the past years, though an extensive survey is beyond the scope of this dissertation.

3.1.3 Search

One way to search for a solution is to exhaustively explore the search space, trying all possible combinations of values. The exhaustive approach is however expensive to perform, and for problems of larger size the approach becomes unrealistic. The standard methods for solving a $CSP(X, D, C)$ are based on (*chronological*) *backtracking search* interleaved with *constraint propagation*.

Backtracking search and constraint propagation

The solution process proceeds by selecting a variable then choosing a value to assign to it. After each assignment, it *propagates* the constraints by removing inconsistent values from the domains of future (or unassigned) variables. If none of the future domains are empty then search continues by selecting another variable; otherwise it *backtracks*, selects another value from the domain of the current variable and continues; if no other values are possible, it *backtracks* to the previous variable. The solving procedure returns as soon as either the first feasible solution is found, or the search has finished finding no feasible solution.

Constraint propagation is very useful to reduce the search effort. However, the balance between level of consistency maintained during search and amount of search involved is very important - tighter levels of consistency can eliminate more inconsistent solutions, but are generally more expensive to maintain. Two popular algorithms for maintaining local consistency in binary CSPs are MAC (*maintaining arc-consistency*) and FC (*forward checking*). MAC, introduced in [84], combines backtracking search and AC enforcement. FC is a restricted version of MAC. Specifically, FC enforces AC only over those constraints involving the current variable and one future (or unassigned) variable. Versions of forward checking for non-binary CSPs are discussed in [15].

In chronological backtracking (CBT), variables are instantiated in chronological order (say X_1, then X_2, .., then X_k, according to the ordering heuristic), and when backtracking is required variables are reconsidered one by one in the reverse order (i.e. X_k, then X_{k-1}, .., then X_1). A different version of backtracking is called *back-jumping* (BJ). This performs backtracking still in the (reverse) original order, but by steps of more variables. Specifically, BJ jumps back from X_k to the first variable $X_{h<k}$ which can lead to a solution - if some reasoning has proven that any re-instantiation of the intermediate variables $(X_{k-1}, X_{k-2}, .., X_{h+1})$ would lead to failure. BJ is then a form of *backward checking*. A particular version of BJ is *conflict-directed* BJ (CBJ), where the back-jumps are established by measuring conflicts among variables. CBJ and hybrid versions of FC-CBJ are discussed in [77]. For hard problems it has been shown that MAC is better than FC (or CBJ-FC) [16].

Variable and value ordering heuristics

Characterizing the solving method requires selecting the type of constraint propagation to be used after each tentative value to variable assignment, but also the variable and value ordering heuristics - the order in which variables and values are assigned during search. The variable and value ordering heuristics define the structure of the *search tree*. Different ordering heuristics can produce trees of significantly different sizes [59], and for which the same type of choice points

$(var_i = val_j)$ can appear shallower or deeper in the search paths.

As the subtree below each choice point is reduced by constraint propagation, and as the amount of pruning can be significantly different for different choice points, the selection of the ordering heuristics can then have a significant impact on the effort required to explore the search tree for a solution. For example, if the search makes a bad choice (or mistake) at the top of the search tree, it can waste a lot of effort exploring sub-trees that have no solution. In [59], the behavior of standard variable ordering heuristics over insoluble sub-trees is compared to optimal refutations, with the advice that some knowledge on how refutations distribute may be relevant to improve the search.

Variable ordering heuristics and value ordering heuristics can be *static*, i.e. orderings are fixed before the search starts, and typically they are chosen to reflect some natural structure of the problem, or *dynamic*, i.e. the next pair (variable, value) is decided online during search, so the choice depends also on the current state of the search.

The standard variable ordering heuristic is dynamic, and is based on the so-called *fail first* principle [86] [87] [6] [8], stating that we should choose the variable with the tightest constraints. The aim is to assign values to variables that are most likely to cause failure as early as possible, rather than later in the search. This is normally implemented by choosing the variable with the smallest remaining domain [53], or the smallest ratio of domain size to the number of constraints acting on the variable, i.e. to the variable's degree of connectivity, representing the CSP as a graph, with variables as nodes and constraints as edges [16]. Other dynamic variable ordering heuristics are *Brelaz* (Bz) and *Kappa* (K) (both examined in [45]). Bz [23] chooses the variable with the smallest domain size and tie breaks on the variable with greatest future degree, while K is based on the concept of *constrainedness* [46], branching on the most constrained variable, which then gives the least constrained subproblem.

A dynamic and adaptive variable ordering heuristics has also been considered in [21]. The heuristic selects the next variable to be assigned picking the one which has provoked the highest number of failures since the search started. The heuristic resulted the best over several abstract and concrete problems.

Strategies aiming to *succeed first* have also been investigated, e.g. in [7] where different variable ordering heuristics showed different search efforts, depending on the level of *promise*.

An example of variable ordering specific for scheduling problems is based on *time-contention* [4]. For example, consider a scheduling problem with a set of tasks to be scheduled on a set of resources: the variables are the start times of the activities, the values are the times to assign, and constraints forbid that pairs of tasks overlapping in time share the same resource. The variable ordering based on contention finds an activity that depends most on the most contended time.

Even the choice of a value ordering heuristic represents an important aspect in setting up a good search algorithm. Among the most effective for many CSPs is the *look-ahead* (or *min-conflicts*) value heuristic [41], which chooses the value that rules out the fewest choices for the neighboring variables in the constraint graph. The general idea for value ordering heuristics (e.g. *min-conflicts*), is to select first the values which are most likely to be successful. However, in some cases, e.g. when all values of each variable must be tried at some point, the fail-first principle (e.g. *max-conflicts*) may be preferred [86].

Standard backtracking versus discrepancy based search

As noted above, chronological backtracking (CBT) is the standard way to explore the search tree. When the subtree below the current assignment is infeasible, the infeasibility may be due to an assignment that occurred earlier in the path coming from the root. Then backtracking occurs, and the most recent assignment of value to variable is retracted. Unfortunately, CBT will exhaustively explore the entire subtree below the wrong assignment before retracting it. Figure 3.1 shows a simple example, where the current assignment ($Z = 0$) is found inconsistent with the previous assignments ($X = 0$) and ($Y = 0$). The real mistake was the assignment ($X = 0$), but CBT will retract the value 0 from X only after completing the exploration of the entire subtree below ($X = 0$).

The computational (or search) waste can be considerable if the wrong assignment is far higher in the search tree. If the ordering heuristic is expected to make few mistakes, then this waste can be reduced by adopting a *discrepancy* based search [5]. The convention is that the left direction in the search tree represents

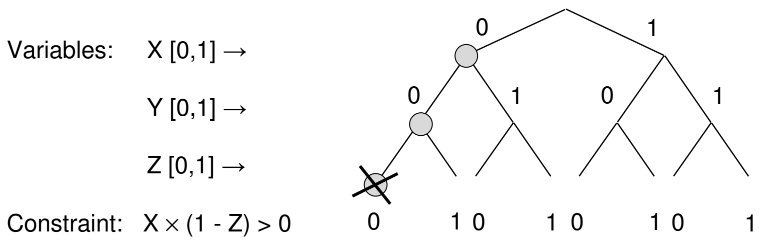


Figure 3.1: Search tree with inconsistency found at the end of the first path explored using CBT.

the heuristic decision. Then, considering the exploration of a search tree from the root node to a solution node, each time we take a right move we go against the heuristic, and we have a *discrepancy*.

Different versions of discrepancy based search have been developed. *Limited discrepancy search* (LDS) [54] explores the search paths by increasing limit of discrepancy (d), i.e. starting with d = 0, then considering $d \leq 1$, then $d \leq 2$, etc. LDS is redundant in the fact that the i-th iteration revisits search paths with fewer discrepancies that have been visited in precedent iterations. *Improved-LDS* (ILDS) [63] starts with d = 0, and then searches over paths with d = 1, then d = 2, etc. This eliminates the redundancy of LDS. *Depth-bounded discrepancy search* (DDS) [94] is based on the assumptions that mistakes are more likely near the top of the search tree than further down. As a pre-iteration, DDS explores the heuristic path (d = 0). Then, the first iteration explores paths on which discrepancies occur within depth 0, the second iteration explores paths on which discrepancies occur within depth 1, etc. CBT, LDS, ILDS, and DDS, are all *complete* search methods, i.e. they eventually explore the entire search tree.

Figure 3.2 represents the same search tree of Figure 3.1. The tree has 8 leaves, i.e. 8 potential solutions. The first line below the tree shows the number of discrepancies occurring in each path root-leaf. The next lines below represent the order leaves are visited, by CBT, ILDS, and DDS respectively. Back to the example of Figure 3.1, note that possible valid solutions are leaf 5 and 7. Then, for the problem represented in that figure, CBT would find a solution (leaf 5) at the fifth path root-leaf explored, ILDS at the fourth path, and DDS at the second path.

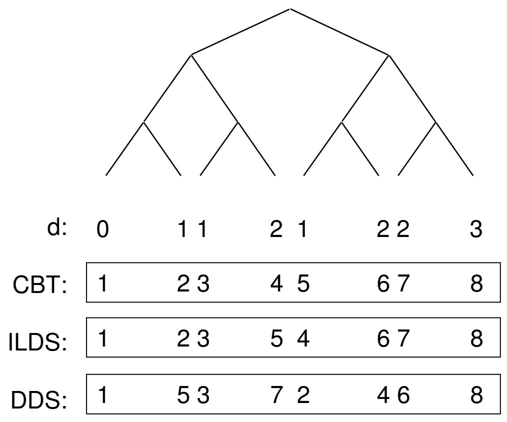


Figure 3.2: Search tree, discrepancy, and order of exploration using CBT, ILDS, and DDS.

Randomized restarts search

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, even if the heuristic is the best on average. Further, if we choose a single randomized heuristic - where tie breaking and value ordering are randomized - and resolve several times the same problem instance, both very long runs and very short runs can occur more frequently than expected [51]. This phenomenon is called *heavy-tailed* [27], and is characterized by a high variance in search behavior of randomized ordering heuristics. In [51] it is shown how the variance can be considerably reduced by using the *randomized restarts* strategy (RR) - for a single ordering heuristic, if no result has been found by a given (and short) time limit, the search is started again. Tie breaking and value ordering are done randomly, and so each restart explores a different path. RR works particularly well on certain problems, including quasi-group with holes. Similarly, *algorithm portfolios* [48] is another randomized restarts method, that searches by interleaving a set of randomized algorithms.

Iterative deepening search

Iterative deepening (ID) is a search technique for general search problems - in Section 4.6 we will develop a search technique based on ID for Constraint Satisfaction Problems. ID is not normally applied to CSPs, where all solutions are at the same depth of the tree. But, in general search problems, solutions could appear at any depths, some requiring a considerably deeper and more expensive search than others. Using *depth-first* search, we may get stuck into a deep area of the search tree where the most expensive solutions are, thus requiring a long time before returning, or perhaps failing to solve, if we are not given enough time. The idea is to search to depth one, and if this first search completes in less than the amount of time allocated then search again to depth two, and if we still have time then search to depth three, and so on, until the time runs out. This procedure looks first for cheaper solutions. ID was introduced in the middle 1970s [64]. The algorithm explores each branch of the search tree to a certain depth, and then increases the depth limit and restarts the search from scratch, repeating this process until either the first solution is found, i.e. the shallowest in the search tree, or the search completes the exploration of the entire tree proving there is no solution.

3.1.4 CSP solubility and phase transition

As discussed previously in the chapter, CSPs are classified in terms of *size* $(n \times m)$, *tightness* (t), and *density* (d). Consider, for simplicity, binary CSPs composed by a set of n variables, each with an initial domain of m values. The average difficulty of solving a randomly generated problem - i.e. a binary CSP where randomly chosen pairs of variables are connected with randomly generated constraints - can be related to the quadruple $\langle n, m, d, t \rangle$.

For example, consider instances of a binary CSP defined by the quadruple $(n = n_0, m = m_0, d = d_0, t)$, where the only variable is t. A value t=0 means that no pair of values is disallowed by any of the binary constraints, so any combination of n values taken from the initial domains is a valid solution. As opposite, when all constraints disallow all possible pairs of values we have the maximum possible tightness $t = t_{max}$, i.e. all pairs of constrained variables have all pairs of values disallowed, so there cannot be any valid solution left. Problems are easier to solve near 0 and t_{max}, and they get harder as t gets away from the two extremes, with a peak of complexity located somewhere in between. For small t, problems are easier because constraints are too loose, so they are more likely to allow a solution, while for t approaching t_{max}, problems are easier because constraints are too tight, so they are more likely to allow no solution. In general, for $t = 0$ 100% of instances are satisfiable, and the percentage decreases as t increases from 0 to t_{max}. The location of the *solubility* transition coincides (over t) with the location of the complexity peak. The sharpness of the transition, the value of the peak, and the location of transition and peak, depend on the other parameters, i.e. n, m, and d. For example, for greater values of density the transition to infeasibility (along with the hardness peak) happens earlier, as more constraints means less chances for a tuple to represent a solution.

When there is a rapid change from all problems having a solution to no problem having a solution, this phenomenon is called *phase transition* [95]. Phase transition has been seen in many abstract NP-complete problems such as 3-SAT, quasi group completion, number partitioning, graph coloring, Latin square, and also in some real problems like job shop scheduling, and sport scheduling.

3.1.5 Extending the representation to improve solving

As defined at the beginning of the chapter, modelling a CSP problem means (i) choosing the *representation*, and (ii) choosing the *solving method*. A single problem can be modelled in many different ways, either in terms of representation or in terms of the solving process. Representing and solving problems can be difficult to do effectively. In the previous sections we have seen how a lot of work has been done in order to make the *solving* part more effective, the main issue being to find the best balance of constraint propagation and search. CP requires skills also in problem *representation*, and knowledge on how solving algorithms interact with representations. In particular, the efficiency of search and constraint propagation can be improved by extending the basic CSP representation with *implied constraints*, *dual modelling*, and *symmetry breaker constraints*.

Implied constraints

The set of constraints in a basic CSP representation are explicitly stated by the problem. *Implied* (or *redundant*) *constraints* [28] [88] are instead constraints that can be derived from existing constraints. They do not forbid any solution allowed by the basic CSP, so they can be added to the model to enforce a stronger propagation during search.

Dual modelling

Often, a problem allows several different CSP representations, depending on which are the decision variables, and on which are the possible values to be assigned. For example, two possible (and *dual*) representations for a scheduling problem are: (i) tasks are variables and resources are values; (ii) resources are variables and values are sets of tasks to be assigned to them. Models based on different CSP representations can have different performances, so a developer should study and compare more versions before implementing the final solution. Further, two representations can often be combined together to achieve better performances. The operation of combination is called *channelling* [56].

Dual modelling has been studied in [56] over *permutation* and *injection* problems, i.e. problems where the number of variables is, respectively, equal to or smaller than the number of values. Note that permutation problems are particularly straight forward to accept dual representations, in fact we only need to exchange variables with values. For example, given three variables, $\{E, F, G\}$, with domains $D_E = D_F = \{H, I\}$ and $D_G = \{H, J\}$, assigning all different values to the variables is a permutation problem. The primal CSP $\langle$X^p,D^p,C$^p\rangle$ is: $X^p = \{E, F, G\}$; $D^p = \{D_E, D_F, D_G\}$; and $C^p = \{(E \neq F) \wedge (E \neq G) \wedge (F \neq G)\}$. The dual CSP $\langle$X^d,D^d,C$^d\rangle$ is: $X^d = \{H, I, J\}$; $D^d = \{D_H, D_I, D_J\}$, with $D_H = \{E, F, G\}$, $D_I = \{E, F\}$, $D_J = \{G\}$; and $C^d = \{(H \neq I) \wedge (H \neq J) \wedge (I \neq J)\}$.

In [56], the authors compare models based on a *primal*, a *dual*, and a *combined* representation. The latter contains both the primal and dual sets of variables, and uses channelling constraints to link the two sets - to maintain consistency. They show that the primal and the dual models are often outperformed by the combined version. In the combined model, the constraints over either the primal or the dual variables are *redundant*, i.e. they can be removed without changing the set of solutions, however results showed that they are useful *implied* constraints to improve constraint propagation.

Symmetry breaker constraints

A CSP can often allow more solutions that are *equivalent*, i.e. for the actual problem, choosing one rather than another makes no difference. A pair of equivalent solutions (say S_1 and S_2) often exhibit some form of symmetry, i.e. $S_2 = f(S_1)$, where f is a 1-1 function such that S is a solution if and only if $f(S)$ is a solution. Symmetries make the search space large and redundant, wasting considerable time during search to explore equivalent assignments. Search effort could be drastically reduced if we could explore only one representative for each group of symmetrical assignments.

Example - Given a CSP containing a pair of variables (X_1, X_2), assigning either $(X_1 = v_1, X_2 = v_2)$ or $(X_1 = v_2, X_2 = v_1)$ can sometimes be equivalent. For example, consider a problem of restaurant table allocation, where X_1 and X_2 are two bookings for 7 o'clock for two parties of size 2, and v_1 and v_2 are two tables of capacity 2. In general, symmetrical assignments can involve tuples of more variables $(X_1, X_2, .., X_n)$. Given the assignment $(X_1 = v_1, X_2 = v_2, .., X_n = v_n)$, there are a maximum of $n!$ symmetries - we have the maximum when all permutations of values $\langle v_1, v_2, .., v_n \rangle$ over $(X_1, X_2, .., X_n)$ are valid and equivalent. For example we can have 10 bookings of size 2 at 7 o'clock (*vars*) and 10 suitable tables (*vals*), which allows up to $10!$ symmetrical assignments.

Symmetry breaker constraints (SBCs) [78] are designed to prune the search space so that, ideally, only one tuple (or partial assignment) from each set of equivalent tuples is considered.

For example, when two variables have identical characteristics, it is pointless to differentiate them ([83], page 37). It is better to model the variables grouping and managing them by type. Similarly ([83], page 151), when two or more constrained variables with identical domains, subject to the same constraints, and being position independent, can be ordered, SBCs can be added to allow only one permutation of the ordering, thus greatly reducing the size of the search tree. For the restaurant example above, with (n=10) identical bookings, a possible symmetry breaker would be $X_i < X_j$ for $i < j$, which would forbid all equivalent permutations but $(X_1 = v_1, X_2 = v_2, .., X_n = v_n)$ - with a considerable reduction in search space, and saving in search effort.

Another variation is represented by SBC for the case where a set of variables can take *indistinguishable values* [44]. For example, in scheduling for restaurant tables, again representing dinners as variables and tables as values, two tables can be identical (e.g. both of capacity two, and both cannot be joined with others), therefore they represent indistinguishable values for any variable assignment.

3.1.6 Constraint optimization problems

CSPs are constraint satisfaction problems, i.e. they are solved by calculating and returning the first solution which satisfies the problem's constraints. Sometimes, however, the same problem can have more possible solutions where some are better than others, according to certain evaluation criteria. A *Constraint Optimization Problem* (COP) is a CSP extended to calculate and return the best of the possible solutions.

$COPs(X, D, C, f)$ are defined as $CSPs(X, D, C)$ extended with an *objective function* f, which maps every possible solution to a numerical value. If S is the set of tuples representing the possible solutions, then $f : S \rightarrow$ *numerical value*. A solution to a COP is then an assignment of values to all the variables which satisfies all the constraints, and with the best (*max* or *min*) objective value.

Solving COPs using branch and bound

One way to solve COPs is to search, going through all possible solutions, and keeping record of the one with the best objective value. An exhaustive search is often too expensive, with a considerable amount of time wasted exploring solutions which do not improve the current best one.

Branch and bound (B&B) is a more efficient approach. B&B constrains the objective value for the next solution to improve over the current best solution. The objective function can be expressed as an auxiliary constrained variable (say $f = X_{Obj}$). If the problem is found with no solutions the algorithm returns. Otherwise, any solution found is used as lower bound for the objective of the next solution. The algorithm proceeds by incrementally assigning values to variables, and after each assignment the lower bound propagates like all the other constraints. If an extension to the current partial assignment reduces the upper bound of X_{Obj}

below the current best value, then search can backtrack. The constraint on X_{Obj} prunes all the subtrees descending from partial assignments that cannot produce any improvement, thus saving a considerable amount of search effort.

Anytime algorithms

In the real world, problems usually require solutions to be available within limited time, e.g. to meet a deadline. The time to find the final (and optimal) solution of an optimization problem could then be unacceptable. *Anytime algorithms* [93] are algorithms that can return solutions of increasing quality at any time over time, eventually reaching the optimal. Algorithms based on B&B are straightforward to provide anytime solutions - in the procedure described above, the current best solution is always available. Good anytime algorithms should however exhibit a proper *anytime profile*, i.e. an increase in quality sharper for earlier times, less sharp as time goes by, and asymptotically null. If the improvement is poor even for the early times, or if the increase levels off too quickly, then the benefit of the anytime solutions may not be worthwhile.

3.1.7 Local search

Many real life applications concern hard and large problems whose search space can be too big to allow a complete search in practical time. As we saw above, one way to tackle hard (optimization) problems is by using anytime algorithms [93], which return solutions of increasing quality over time. However, if these algorithm are still based on a complete and systematic search, the complexity of the problem may not allow a proper anytime profile.

Local search (LS) represents a different approach for solving computationally hard problems, either of satisfaction or of optimization. Standard LS starts by generating a candidate solution, based on heuristics or randomly, which can be infeasible, sub-optimal, or incomplete. The method then iteratively applies minor changes to the initial solution, aiming to improve it, i.e. to achieve a feasible, optimal, or complete solution.

There are different LS approaches, where the difference is characterized by the criterion by which changes are applied, by the type of improvements sought, and by the type of reaction performed when search gets stuck in a local unsatisfactory point. Among the most popular algorithms based on LS we mention hill-climbing (HC), genetic algorithms (GA), tabu search (TS), and simulated annealing (SA). An overview to these and other approaches can be found in [58].

HC attempts to maximize an objective function f: local changes are accepted only if they make the value of f increase, and the process continues until a (local) maximum is reached. GA maintains a set of solutions, and the procedure finds improved solutions by imitating the natural evolution process: solutions are combined or mutated to change the current set, and those of poor quality are penalized or eliminated. TS maintains a tabu list of partial or complete solutions, i.e. visited search paths which cannot be revisited, and explores the solution space by changing the initial solution, aiming to move to a better solution. SA explores the search space by generating neighboring solutions of the current solution, and accepting the new solutions probabilistically, depending on the quality achieved, and on a parameter called *temperature* (used to modify the search process).

Any pure LS method cannot guarantee a feasible, optimal, or complete solution. The main problem of LS algorithms is that they can get stuck in points of unsatisfiability - e.g. local optimum, if we are solving an optimization problem. The common strategy adopted to escape from these points is based on randomizing the search process, i.e. performing random moves around the current solution. Combined to that, each LS method has its own tactic to further protect the search from local stagnation. For example, when the improvement process deteriorates: HC can make perturbation to the current zone of search by temporarily allowing down-hill movements; GA can make perturbations by changing the cross-over, mutation, or penalization functions; TS exploits its tabu list to avoid repeating the same paths; and SA varies the temperature parameter, thus increasing the probability of acceptance for poorer solutions, and therefore allowing to visit more neighbors in the search space.

Local and complete search can be combined within a constraint programming framework to get the benefit from the three sides [58]. First, both types of search get the benefit of constraint propagation, which prunes the search space after each decision point. Second, local search can be used when a sub-optimal solution is needed within a limited time. Third, solution optimality can be available when wanted if we allow local search to switch to complete methods.

3.1.8 Variations to standard CSPs

Often, in reality, some constraints are not strict, some are simply preferences and can be violated, some are more important than others, and some are uncertain. One limitation of standard CSPs is that they cannot express preferences or uncertainty among constraints: constraints are either required or ignored (i.e. not included). Basic CSPs have been extended in several ways to allow a better representation of the real world.

Partial CSPs (or PCSPs) [40] allows constraints to be violated, aiming to minimize the number of violations. *Weighted* CSPs (WCSPs) [40] extend PCSPs by assigning a weight to each tuple and then minimizing the weighted number of violations. *Max* CSPs [75] are WCSPs where tuples can have weights of value 0 or 1, so the aim becomes to find a solution with the minimum number of violated tuples.

Another popular version, designed to handle preferences, is called *Fuzzy* CSPs (FCSPs) [82]. Here, each tuple is assigned a ranked degree of satisfaction (or membership function) normalized between 0 and 1. The goal is then to maximize a combination of the satisfaction degrees of all tuples in a solution - e.g. typically the aim is to maximize the minimum degree over all the constraints. FCSPs lend themselves well to many real problems, where it is difficult to precisely elicit numerical data in terms of constraints, probabilities, costs, objectives, etc.

Mixed CSPs [37] are an attempt to model uncertainty. The set of variables is split into controllable decision variables and non-controllable parameters. The solver can assign a value only to the controllable variables, while the values of parameters depend on external sources and are not known in advance. A complete assignment of the parameters is called a *world*, while a complete assignment of the decision variables is called *decision*. If the values of all the parameters are revealed before the decision deadline we have *full observability*. In this situation, solving a Mixed CSP means computing an approximate (*conditional*) decision, i.e. a decision that associates different assignments of values to different possible worlds. If none of the parameters are revealed before the decision deadline we have *no observability*. In this situation, solving a Mixed CSP means computing a pure (*unconditional*) decision, i.e. a decision that assigns values to all the decision variables aiming to satisfy as many worlds as possible. As defined in [92] [72], a problem is: *strongly* controllable, if there exists a single decision that is a valid solution for any possible world; *weakly* controllable, if there exists a decision for each possible world; and *dynamically* controllable, if, assuming the world is partially unknown at decision time, there exists a partial decision based on the observation of the known part of the world that is ensured to extend to a complete decision (and solution) whatever parameters remain to be revealed. Strong controllability is suitable in case of no observability, weak controllability in case of full observability, and dynamic controllability in case of partial observability. In particular, dynamic controllability suits well dynamic application domains such as planning or scheduling, for which the solution is built incrementally, over time, as the uncontrollable parameters reveal their values.

In [18] *semiring* and *valued* CSPs are introduced as general frameworks, which allow to represent different types of CSP variations, among them (standard) CSPs, PCSPs, WCSPs, Max-CSPs, Fuzzy-CSPs, and Mixed-CSPs.

3.1.9 Dynamic problems: modelling changes and uncertainty

Dynamic problems are problems that change while they are being solved, or as the solution is being executed. Many potential applications of constraint programming turn out to be dynamic problems [24], for example, dynamic scheduling [29]. In scheduling, a machine may break down, or a scheduled action may be delayed due to the late arrival of supplies. In general, a dynamic problem needs to be solved online, i.e. partial solutions must be generated and executed to accommodate the changes, before the complete problem is known.

In constraint programming terms, the problem is to decide which values to assign to variables before all the variables and constraints are known. Further, if we have some knowledge of the possible future developments of the problem, we should try to use that knowledge to make our initial assignments. Many techniques from the literature are relevant to tackle different aspects of dynamic problem solving. The main contributions are presented next.

Probabilistic and stochastic CSPs

Probabilistic CSPs [36] (PrCSPs) and *Stochastic* CSPs [67] (StCSPs) extend Mixed CSPs by modelling the probability distribution of the possible values each uncontrollable variable (or parameter) can take.

PrCSPs solve the decision problem by maximizing the probability over all the parameters. For example, consider the problem with two decision variables X_1 and X_2, and two parameters P_1 and P_2, all with domain {0,1}, and the constraint $X_1 + X_2 + P_1 + P_2 = 3$. There are four possible assignments, i.e. the pair (X_1, X_2) can take values (0,0), (0,1), (1,0), and (1,1). Consider the probability distributions over the values of P_1: {0 : 0.2, 1 : 0.8} and P_2 : {0 : 0.6, 1 : 0.4}. Then, the four possible realizations for the pair of parameters $\langle P_1, P_2\rangle$ have probability: {(0,0) : 0.12, (0,1) : 0.08, (1,0) : 0.48, (1,1) : 0.32}. Assignment (X_1, X_2) = (0,0) has probability 0 to satisfy the constraint, (X_1, X_2) = (0,1) has 0.32, (X_1, X_2) = (1,0) has 0.48, and (X_1, X_2) = (1,1) has 0.56 (i.e. 0.08+0.48), so the latter assignment would be the best decision.

While PrCSPs maximizes probability, StCSPs look for solutions with a combined probability greater than a certain threshold θ. Further, while PrCSPs reason on probabilities over a single stage, StCSPs extend the reasoning to multiple stages. Specifically, the problem is solved by deciding the values to the controllable variables in the first stage, then the values of a first set of uncontrollable variables are revealed, then deciding the values of the controllable variables in the next stage, then the values of the next set of uncontrollable variables are revealed, and so on.

Both PrCSPs and StCSPs are suitable for problems which allow an accurate model of probability. Considering the problem of restaurant table allocation (Chapter 2), future knowledge is often approximative or unavailable - most restaurants do not store any form of data necessary to elaborate good probability distributions on table demand (i.e. nature and number of new requests, cancellations, and booking changes) and delays (i.e. late/early arrivals, and expected dinner durations). Further, modelling using StCSPs involves multiple stages, with assignments to variables on one stage depending on previous assignment of variables in the previous stage. This is quite unrealistic for the restaurant problem, e.g. the duration of a dinner early in the night is hard to relate to the duration of another dinner later in the night. In general, the restaurant problem is dynamic, has many sources of uncertainty, and requires online decisions. Representing this type of problems using PrCSPs, and (especially) StCSPs, can be too complex and expensive to do effectively, requiring the generation of a probability tree of very large size, and perhaps an online redefinition of the tree each time an unexpected change happens.

Branching CSPs

Another approach used to tackle problems where we have some model of future changes is *Branching* CSP (BrCSP) [38]. BrCSPs model problems that grow over time, and for which new decision variables arrive as the current set is being assigned. To assign a newly arrived variable, the BrCSP approach searches and propagates constraints over a tree of possible futures (*probabilistic tree*). Each path in the tree is a possible and distinct sequence of arrivals (future decision variables), and each variable in a sequence has a probability of arrival which depends on the previous variables in the sequence. When a variable arrives it can be assigned a value or it can be rejected. There is a specific *utility* associated to the assignment of each variable, while rejected variables provide no utility. A solution to a BrCSP is then an assignment of values to variables which maximizes the utility over the probabilistic tree, where the utility of each assignment is weighted by the associated probability.

BrCSPs are suitable to model problems whose future development can be represented using probabilistic trees, e.g. a scheduling problem where the arrival of a task later in time can be related to the arrival of a task arrived earlier. For the restaurant allocation problem we do not have such type of relations, as parties are essentially independent variables (although there may be some overall booking patterns).

Sampling methods

Even when we have a precise knowledge of the future distribution of a problem, maximizing the expected utility over the entire tree of possible futures may require unrealistic time. Bent and Van Hentenryck [10] [11] [12] [26] propose approximate methods based on sampling a number of future developments and on optimizing the current decision over the samples. Given enough time, the set of samples can still be extended until all the future combinations can be explored and evaluated.

A first version is based on *expectation* [26]. The current variable to be decided (X) is assigned, in turn, each possible value in its domain (D_X). For each assignment, the problem is solved over the set of future samples, and the expected utility of each solution is counted. The value showing the greatest utility over all the samples is then the final decision.

A second version is based on *consensus* [11] [12]. Here, the problem is immediately solved over the set of future samples. Then, considering the current variable to be decided (X), the value that was assigned most to X over all the solutions is finally taken. If m is the number of values in the domain of X, and n is the number of samples, then *expectation* performs $m \times n$ optimizations, while *consensus* does only n, i.e. the latter is cheaper but also less accurate.

A third version is based on *regret* [10], and is a compromise between the previous two. Given a decision $X = a$, the regret of not using another available decision $X = b$ is defined as the difference in objective value between the solution obtained with $X = a$ and the alternative solution with $X = b$. Here, the problem is immediately solved over each sample (as for consensus), but for each sample solution, all the alternative decisions are evaluated in terms of regret. The procedure finally selects the decision which minimizes regret over all the samples. This third version performs the same number of optimizations as *consensus*, however the extra work to compute regrets improves the solution quality.

Sampling has proven to be a successful approach for different dynamic problem domains, for example packet communication scheduling [10] [11] [26], dynamic vehicle routing [10] [12], and on-line multi-choice knapsack (or holiday reservation scheduling) [9]. In these cases, the type of stochastic variable to consider concerns the arrival of future packets, for packet scheduling, of future customer orders, for vehicle routing, and of future items, for the knapsack problem (or of future requests for holiday packages, in reservation scheduling).

Sampling could also be applied to tackle the restaurant table allocation problem. For example, it could be used to decide, at each point during a dinner session, the best seating plan, that is the allocation of the current set of parties which maximizes the chances to accommodate future parties, i.e. to satisfy future meal requests. In this case, the stochastic variable would be the arrival of future meal requests (or table demand).

Assuming we have a way for generating good samples (e.g. based on past booking sheets), the problem may require a large number of samples, in order to get a significant estimate of utility. Even if we choose the cheaper version based on consensus, the amount of time required to evaluate the samples may represent an issue. The complexity of the resolution increases with the number of samples, but also with the sample size. For example, for a restaurant with a target turnover of 100 parties, samples of up to 100 parties may be necessary, for which optimization might become too complex to do effectively. Provided we can be fast enough to perform optimization, sampling could represent a valuable approach for restaurant table management.

Dynamic CSPs

Dynamic CSPs [34] model a sequence of changes as a sequence of CSPs. Each CSP is obtained by adding or removing constraints to the previous CSP. The aim may be to minimize the effort to find new solutions, or to minimize the distance between successive solutions - large differences between successive solutions are often undesirable in real world problems.

The approach in [85] consists of recording *nogoods* during search, for reuse in future solutions. In [91], the authors present an algorithm which tries to accommodate each change by performing *local changes* starting from the solution to the preceding CSP in the sequence. Specifically, the set of decision variables are partitioned into three subsets: in the first variables have fixed assignments; in the second they have assignments which can be modified; and in the third they are unassigned. When the algorithm starts, all variables are in the second set, and each variable is assigned the same value taken in the previous solution (before the change). If the initial assignments can accommodate the change then we do not need to search, as no constraint is violated. Otherwise, at least one variable in the scope of each violated constraint is selected and moved to the set of unassigned (the selection is heuristic). To repair the conflict, unassigned variables are then recursively reassigned different values from their domains. If a reassignment to a variable does not satisfy all the constraints, the new value is fixed and the variable is moved to the set of fixed. In this case, search proceeds moving more variables from the assigned to the unassigned set, so that the problem is made consistent with the current fixed assignments, and then (again, recursively) selecting and reassigning another variable in the unassigned set. If search proves that the problem has no solution for a certain value given to a fixed variable, it backtracks and tries with another value.

Depending on the heuristic order in which variables are unassigned, reassigned, and fixed, the same problem instance may produce different solutions. This technique does not guarantee optimal stability, i.e. the new solution may not be the nearest possible to the preceding one. Considering the problem of restaurant table allocation, it is possible that, for example, to accommodate a delay, *local changes* would return a solution which requires the reallocation of a considerable number of tables, even though there was another possible solution with only a few changes. Restaurant staff and customers may get annoyed and confused when several seating disruptions occurs, especially if this requires many reconfigurations of tables.

Two approaches which appear more suitable in order to control stability in the restaurant problem are [74] and [79]. Petcu *et al* [74] propose special stability constraints, which must be satisfied in order for the solution to be stable. Alternatively, Ran *et al* [79] search for optimally stable solutions. Representing a dynamic CSP as a sequence of CSPs, the latter approach looks for a solution to the current CSP that has the minimum number of different assignments compared to the solution of the CSP immediately before in the sequence. The authors describe a repair-based algorithm with arc-consistency (RBAC) which combines local search (based on iterative deepening) with constraint propagation. Specifically, RBAC starts with the original solution and iteratively checks whether reassigning one variable, two variables, etc., is sufficient to solve the new problem.

The RBAC algorithm is efficient to find the minimal change solution for problems with unary constraints, however the performance deteriorates for problems with constraints of greater arity. In [79], the same authors propose two new algorithms for near optimal solutions. The first, BS, performs binary search, varying the limit on the number of variables that can change value (i.e. the search depth), and applying max-conflict variable ordering selection, and min-conflict value ordering selection. The second, RS, is based on depth first search, setting the depth limit to the total number of variables (max depth) or to the depth of the best solution found so far (called search-depth adaptation), and combines search restart with a randomized variable selection. Both BS and RS limit the number of backtracks over each search step. Results show how the two algorithms represent a reasonable trade off between efficiency and solution optimality. In general, the best performance is achieved using RS, setting a large number of restarts, a moderate limit on the number of backtracks between restarts, min-conflict value ordering selection, and search-depth adaptation.

Temporal robustness in scheduling problems

Some approaches aim to prevent instability by providing robust solutions, i.e. solutions that are likely to remain solutions when changes occur, or that can be modified with little disruption. In [30] robust solutions to scheduling problems are achieved by adding *slack* to activity durations. The authors consider three versions. In the first, the duration of each task $t_i.dur$ is extended to $t_i.dur{+}slack(t_i)$, i.e. the end of the task is right shifted. In the second, each task maintains its original duration, but constraints are modified to require specific *slack* after each task in the schedule. The third case is similar to the second, with the difference that now we have *focused* slack, depending on the temporal location. For example, [22] shows the benefit of penalizing early idle times (which are early slack) in dynamic job shop scheduling.

Using slack to protect schedules from disruptions could be effective also for scheduling restaurant tables. For example, a restaurant may find itself systematically late in providing food on time to all the customers during the peak hour (say 9 o'clock). Several tables are then freed later than expected, and this causes seating plan disruptions, and delays for future diners. As we do not known in advance which tables are going to be late, a simple solution to reduce disruptions (and delays) could be, for example, to add some slack on a subset of tables at around 9 o'clock. Figure 3.3 shows a small seating plan with 3 tables, 6 parties, and a slack time (deliberately) inserted on table $T3$ in correspondence of 9 o'clock. The vertical (dashed) line represents the current time, where party $P2$ is expected to finish. Thanks to the slack time on table $T3$, if $P2$ stays longer than expected the delay can be absorbed with a minimum disruption (and with no extra delay), by swapping $P5$ with $P4$.

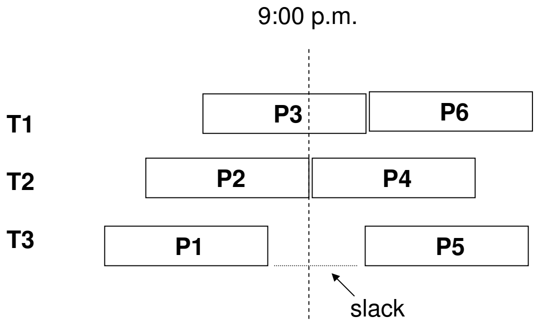


Figure 3.3: Example of temporal robustness increased using slack for the table allocation problem - if $P2$ finishes late, $P5$ and $P4$ can swap without introducing any more delay.

The main issue in arranging slack protections would be to find out how much slack to add, and where. Customer behavior and restaurant performance are the main causes of delays. The high level of uncertainty governing these two factors may not allow an accurate and effective choice of slack times, and over-estimates can lead to a considerable waste in table usage and profit.

An alternative approach to tackle problems of scheduling with uncertain durations is represented by *Just In Case* scheduling (JICS) [35]. JICS attempts to anticipate likely changes in task durations that could cause schedule breakage by pre-computing contingent schedules. Specifically, JICS identifies the most likely breakage point in an initial schedule, computes an alternative schedule from that point on (accommodating the breakage), and given more time, repeats the process, computing further alternative schedules accommodating the next most likely breakages. Note that the set of contingent schedules is built so that it can at the same time manage breakages and preserve resource usage, whereas the introduction of slack handles breakages but degrades resource usage. The main issue in

JICS regards the uncertainty of the breakages. For example, in restaurant table allocation, unexpected events which can cause seating plan disruptions (and delays) are very frequent and uncertain.

Consider a seating plan, where each party has a size and a start time, and is assigned a dinner slot of a certain (expected) duration on a table of suitable capacity. Each party can generate 3 types of unexpected events that can potentially create conflicts in the seating plan: (i) the party may arrive in a larger group than the size of the booking, so the assigned table may no longer be suitable; (ii) the party may arrive late, so there may be a clash with the next party planned on the same table; (iii) the party may occupy the table for longer than expected, so again, there can be a conflict with a following party. Considering that each party can have any of the three changes (in size, start, and duration), the combinations of possible breakages over a dinner session makes the implementation of contingent schedules unrealistic.

Super solutions

A more general framework for solution robustness, which can be applied to any problem domain involving changes, is represented by *Super solutions* (SS) [55]. SSs are solutions that guarantee a limited number of repairs in case of changes. Formally, a solution is (a, b)-SS if for any a variables which lose their values (called the *break set*) there is a reassignment of at most b other variables (*repair set*) which repairs the solution. A variation to (a, b)-SS is represented by (α, β)-SS [57]. An (α, β)-SS is a solution which guarantees that for each combination of assignments to variables with a total probability α of being lost, the solution can be repaired by reassigning any subset of variables at a total repair cost less than β.

In (a, b)-SS, the value b characterizes the *stability* associated to the super solution, i.e. a smaller size of the repair set means that the solution can accommodate any change of size a by rearranging fewer variables, so maintaining a higher stability. The problem of finding (a, b)-SS is NP-complete for any fixed a. In [55], the authors focus on algorithms to calculate $(1, 0)$-SS, i.e. solutions for which any single change can be repaired without reallocating any other variable, thus allowing the maximum stability. When there is no $(1, 0)$-SS, the choice is then to maximize the number of variables which can be repaired with no changes. For example, a $(1, 0)$-SS for the restaurant allocation problem can be a seating plan which allows any party to finish late (e.g. by 15 minutes) without the need to reallocate any other party - i.e. there must be a slack of at least 15 minutes after each dinner slot. If no such solution exists, then we may search for one with the maximum number of dinner slots followed by a 15 minute slack.

In the example above, a $(1, 0)$-SS protects from single delays of 15 minutes over any dinner duration. However, changes are very frequent and diverse in the real situation. The same dinner can undergo multiple and profound changes, for example a party of 8 may turn up at 6.30 p.m., even though their booking was made for 6 people at 6 o'clock, and then the same party may occupy the table for 3 hours, even though the expected duration was 2 hours. Therefore, the super solution approach may become too complex if we have to represent all the possible forms of changes. Further, if SS_0 is an (a, b) super solution and S_0 is any non-super solution to the same initial problem instance P_0, then SS_0 may have more chances than S_0 to repair the first change C_1. However, assuming both SS_0 and S_0 allow a repair for C_1, the super solution approach does not guarantee that the solution obtained after repairing SS_0 is not now poorer in robustness than the solution obtained repairing S_0. In other words, the current version of (a, b)-SSs may not be suitable to model problems like the restaurant one, which requires to accommodate chains of changes.

Ideally, to control robustness over chains of changes, a restaurant should aim for an initial seating plan which represents a recursive form of SS (RSS), e.g. we can call it $(a, b)^n$-SS. A $(a, b)^n$-SS would be a solution that, for any a variables that lose their values, can guarantee a new solution through a repair of size b, where the new solution is then $(a, b)^{(n-1)}$-SS. The value n represents the number of consecutive breakages to the $(a, b)^n$-SS for which a repair is guaranteed. Typically, in the restaurant problem, changes happen one by one during the night, so we could use the relatively cheaper form $(1, b)^n$-SS. This idea of RSS, even considering a = 1 and n small, appears however unrealistic to implement, in terms of complexity. Further, we expect that RSS would degrade the restaurant load, protecting each party using consistent slack times and perhaps leaving some tables unoccupied.

3.2 Description of scheduling

Scheduling is the process of allocating tasks to resources over time. Each scheduling problem that may be considered has got its own configuration in terms of task and resource characteristics, and metrics of performance [76] [3].

Each task is typically defined by: a *release* time, the time by which it becomes ready to start the execution; a *due* time, the time by which the execution must be completed; a *start* time, the actual time by which the execution starts; an *end* time, the actual time by which the execution is completed; a *duration*, the time required to execute, where *duration* = *(end - start)* if the execution cannot be interrupted; and a *value*, which usually reflects a ranked priority, importance, profit, or cost. Different tasks may have different subsets of resources where they can be processed. Tasks can be individuals and independent from each other, otherwise they could come as ordered packages.

The resources can be one or more, some may process only a single task at a time, others may process more tasks simultaneously. The resources can be all identical, otherwise the may have different capacities, or each task could take time dependent on each resource.

In *decision* problems, the responsibility of a scheduler is determining whether a schedule exists, i.e. whether all tasks can be allocated such that all timing constraints and resource constraints are satisfied. Often time constraints cannot be all satisfied, e.g. we may need to reject some tasks, or to allow some tasks to complete late. What happens when timing constraints are not met depends on the type of application. For example, an online system that controls a nuclear power plant, of course cannot afford to miss timing constraints of the critical tasks. In restaurant allocation, instead, a violation of a time constraint can be caused by a dinner lasting longer than expected and causing an overlap with the next dinner scheduled on the same table, the effect being that some future diners will have to wait before getting seated. In general, in *optimization* problems, the scheduler must maximize some performance based on *objective functions*, which are usually a balance of value, completion time, or lateness.

3.2.1 Dynamic scheduling

When the set of tasks to be allocated changes as the solution is being executed we have *dynamic* or *online scheduling* [47]. Many real world scheduling problems turn out to be dynamic, for example:

- manufacturing scheduling - new orders arrive, and must be integrated into existing schedules;
- hospital scheduling - new patients arrive in a more or less random fashion, with problems of different importance and urgency, and must be assigned to beds, theaters and staff;
- delivery scheduling - new orders arrive at random times, and must be assigned to couriers;
- reservation scheduling (e.g. car rental, holiday booking, or restaurant table allocation) - new requests arrive at random times and must be committed to resources, where each request concerns an activity (e.g. rental, holiday, or dinner) which is typically assigned a fixed start time and duration.

The real world is governed by *uncertainty* not only on the arrival distribution of future tasks, but there may be no complete knowledge also on the current set of tasks to be executed: resources are not available when required, current tasks get modified or cancelled, some take longer than expected to complete, others cannot start on time. For example, in restaurant table allocation (Chapter 2), tables are resources and dinners are tasks, and common (unexpected) changes can be new bookings, or new parties arriving without a booking, but also late or early arrivals, dinners lasting longer or shorter than expected, cancellations, or booking modifications. In general, the aim is to provide a certain level of service, despite the changes. In particular, the problems may have different objectives, including ensuring every task can be carried out, minimizing delay, minimizing reallocation, or maximizing importance.

A large variety of scheduling problems has been formally classified and defined in the literature. An excellent overview on scheduling applications can be found in [76], while [29] provides a survey of techniques for scheduling with uncertainty.

3.2.2 Solving dynamic scheduling

In static scheduling, the complete set of tasks to be scheduled is known in advance. In dynamic scheduling, the set of known tasks changes over time - new tasks may arrive, others may get cancelled, others may change either start time, duration, or resource requirement - so the scheduler has to react each time a change happens, without knowing the future changes, and the final number and nature of tasks. Due to the presence of uncertainty, no strategy can guarantee an optimal solution for the dynamic problem [89]. For example, determining a non-preemptive schedule, that does not permit the removal of tasks from a resource as a higher priority task arrives, is an NP-hard problem even on a single resource when tasks can have arbitrary release times [42]. In [71], they studied multi-resource online scheduling problems, noting that with such problems no algorithm is optimal and can guarantee all tasks without prior knowledge of tasks arrival, processing, and due times. This knowledge is not available in dynamic systems, so it is necessary to resort to approximate algorithms (or heuristics) to construct the schedules. Heuristic strategies can be more or less complex, but do not involve a complete analysis of the problem. Many systems focus on the use of *heuristic* strategies which assign priorities to tasks. The aim is typically related to the objective functions listed above. For example, very simple and popular heuristics for dynamic scheduling are FCFS (i.e. first come first served) and EDF (earliest deadline first) [89]. Both FCFS and EDF are cheap and greedy attempts to optimize a function of task value and "earliness" (or delay). More complex heuristic strategies will be discussed below.

Solution stability

In dynamic scheduling, *stability* is another popular objective that often needs to be balanced along with *value* and *delays*. Consider the example where we are in a train station, waiting for our train (e.g. a regional) to arrive at the assigned platform, but then the speaker announces a change in platform, and perhaps a delay, for the departure. This change may have been caused by another train (e.g. a Eurostar) with a higher priority and value being late. The decision on whether to delay some lower priority trains to let the Eurostar recover, and in case, on which trains to delay and for how much, is a matter of maximizing *value*, minimizing *delays*, but also maximizing (or ensuring) *stability*. In fact, the best decision for value and delays may require the reallocation of the departure platform of many trains. This, however, may not be acceptable or reasonable. More passengers having to move from one to a new platform means that more people get annoyed, and more risk to miss the train because they have not noticed the change in time. Further, the flows of people crossing the station and triggered by the change may cause unpleasant and unsafe situations, as well as further delays.

Case study example: metrics for restaurant table allocation

Value, delay, and stability objectives are very relevant also to restaurant table management. In this dissertation we have studied the problem of restaurant table allocation (RTA) as a scheduling problem with tables as resources and parties (or dinners) as tasks. RTA is dynamic, as the set of parties changes over time - new parties may arrive, others may get cancelled, others may change start time, duration, or size (i.e. table requirement) - so we need to reschedule each time a change happens, without knowing the future changes. Typically, the restaurant profit is proportional to the turnover of people (or *total covers*). Each party size then represents the value of the party, and when a new party cannot be scheduled it is rejected, gaining no value.

Possible metrics for RTA could then be based on:

1. *value*, for which the best scheduler is one that maximizes the total value of scheduled tasks over the considered time period (i.e. a dining session);

2. *robustness*, for which the best scheduler is one that, over the dining session, accommodates more changes (e.g. delays in party start time or duration) without delaying other parties, or minimizes delay, e.g. the sum of waiting times over all parties;

3. *stability*, for which the best scheduler is one that minimizes the number of table reallocations required to accommodate changes, over the dining session - in fact, frequent table reconfigurations or reallocations of pending dinners can create confusion and noise for both customers and staff.

Ultimately, restaurants must trade off among value, robustness (or delay), and stability. Note that, heuristic strategies purely based on any one of the three metrics may be unreasonable, i.e.:

1. maximizing value would try to fit in as many people as possible, but this increases the risk of both delays and seating plan instability;

2. maximizing robustness would accommodate changes, aiming for a seating plan reallocation that minimizes delays (according to some metric), but this may require many seating disruptions and may as well degrade resource usage and therefore potential value - e.g. moving many parties, and many of small size into oversized tables;

3. minimizing instability would react to changes looking for minimal reallocation, which may not be optimal in terms of both value and delay - in fact, in the order, a local reallocation does not guarantee optimal (or even good) resource usage or minimum (or even acceptable) delays.

Reactive and proactive scheduling

Dynamic scheduling under uncertainty can be approached using either proactive or reactive scheduling models [29].

Proactive scheduling models (PSMs) predict future changes by reasoning about statistical knowledge of uncertainty, and so they compute *robust* schedules, which are more likely to remain valid after a sequence of changes, or which require cheap repairs in case some violations may happen. The computation can be quite expensive, depending on the level of statistical reasoning. Therefore, PSMs (typically) operate *off-line*, sometime before the schedule execution time, and with a good amount of time available for computation.

PSMs can spend a lot of effort to provide robustness in absorbing uncertainty. However, in many practical situations, the environment can be quite unstable, and the frequent changes require fast reactions. Further, even when the environment changes less rapidly, PSMs cannot always take into consideration all sources of uncertainty, so unpredicted events can still cause schedule breakage, which requires a real-time repair. Because of these issues, many practical cases rely on cheaper and faster scheduling models (called *reactive*) or on a combination of a *proactive* phase (off-line) followed by a *reactive* one (on-line).

Reactive scheduling models (RSMs) are perhaps less clever, but generally cheaper and faster than PSMs, as they do not reason about possible future developments of the problem. RSMs may either regenerate a completely new schedule each time an unexpected event happens (e.g. [25]), or they may simply reuse the solution before the change as a starting point for a new solution (e.g. [91]). RSMs operate *on-line*, at execution time, when the information about the state of the scheduling problem is up to date, but the time available to solve can be short.

Different approaches for modelling changes and uncertainty in (generic) dynamic problems have been presented in Section 3.1.9, where we have seen (in particular) how they can also be applied to the specific problem domain of dynamic scheduling. Some of such methods can be regarded as proactive, others as reactive, and all focus on optimizing a combination of some metrics of quality, robustness, and stability.

For instance, the CSP based models of *Probabilistic* [36], *Stochastic* [67], and *Branching*[38] CSPs are proactive approaches founded on probabilistic models of the uncertainty. Their focus is more on maximizing some intrinsic quality of the problem rather than robustness or stability. The methods based on sampling [10] [11] [10] [12] [26], such as *expectation*, *consensus*, and *regret*, are a hybrid between reactive and proactive approaches - they react to changes making decisions online, but their decision is based on proactive sampling of possible future developments. Again, the scope these techniques have been tested for concerns maximizing quality. *Dynamic* CSPs [34] have also been tackled using reactive methods [91] [79] [74], where solutions are reused, and there is no reasoning on possible futures. The main objective is a balance of quality and stability. The methods have been applied for example to on-line satellite scheduling [91]. Adding *slack protections* to schedules [30] is proactive, as it takes into account uncertainty in task duration when forming the initial schedule. Similarly, *just-in-case* scheduling

(JICS) [35] is also proactive, building contingent schedules to accommodate the most likely temporal breakages. Both the methods aim for schedule robustness, and JICS also for quality. Finally, *super solutions* (SS) [55] are again a proactive method, which guarantees a limited number of repairs in case a number of variables may break. In this case, the breaks can be more general than just temporal, e.g. they can be caused by a task completing late, but also by a task changing the resource requirement - e.g. in the restaurant problem, when there is a six-seater table booked for a party of 6 people, but then the party arrives in a group of 8 people. The main characteristic of SS is robustness and stability.

3.2.3 Scheduling jobs with fixed start and end times

Scheduling jobs with fixed start and end times (S_{FSE}) [2] is a subclass of job shop scheduling [42], and represents the base for our scheduling model for the restaurant table allocation problem. Specifically, in S_{FSE} each job J_i is a single task, has a value v_i, a fixed start time s_i, and a fixed end time e_i (i.e. there is no slack).[†] In S_{FSE} there are no precedence constraints between jobs, the only timing constraints being those to respect the start and end times, while the resource constraints depend on the type of resources, i.e. they ensure that jobs go into allowed machines.

S_{FSE} with identical machines

In [2], the authors consider a first simplified subproblem where all m machines are identical, with each machine being able to execute each job. The problem is then to find a subset of jobs that allows a feasible schedule, and that maximizes the value of jobs to be scheduled. They reformulate the problem in terms of *m-coloring* over *interval graphs*.

[†]The problem can also be regarded as the "Reservations without Slack" problem [76].

m-coloring on interval graphs - An *interval graph* (IG) is the intersection graph $G(V, E)$ of a set of intervals over the real line - i.e. each vertex in V is an interval over the real line, and each edge in E connects vertexes corresponding to intervals overlapping in time. A *coloring* of a graph G is a function $f : V \rightarrow \mathbb{N}$ such that $f(v) \neq f(w)$ whenever $(v, w) \in E$. An *m-coloring* is a coloring f such that $f(v) \leq m$ for every $v \in V$. The m-coloring (or *vertex*-coloring) problem takes as input a graph G and a natural number m, and consists in deciding whether G is m-colorable or not. The m-coloring problem is *polynomially solvable* for many classes of graph, e.g. IG [52].

In [2], the authors represent each job as an interval whose end points are specified by the start and end times of the job. Each interval has a value (that of the corresponding job), and any interval can take any color from a common set of m (representing the set of machines, where each machine can execute each job). The objective is a variation of classical m-coloring, and consists in maximizing the value of the subset of intervals legally colored. The problem is still polynomial. They describe an algorithm that runs in $O(nlogn)$, with n number of jobs.

S_{FSE} with identical machines and preassigned jobs

A subclass of S_{FSE} with identical machines considers that a subset of jobs are preassigned, i.e. the problem consists in completing a partial schedule. This problem arises, for example, when an existing schedule has to be reviewed, where some jobs have already started execution and cannot be reallocated (or preempted). The problem can be modelled as a *precoloring extension* problem on *interval graphs*.

Precoloring extension on interval graphs - The *precoloring extension* problem [20] is a more general case of m-coloring, where a vertex subset is colored, and the goal is to extend this partial coloring to a valid m-coloring of the whole graph. The problem takes as input a graph $G(V,E)$, a subset $W \subseteq V$, a coloring f' of W, and a natural number m, and consists in deciding whether or not G admits an m-coloring f such that $f(v) = f'(v)$ for every $v \in W$. The precoloring extension problem on IG is NP-complete [68].

S_{FSE} with non-identical machines

In [2], S_{FSE} is extended to the case in which machines are no longer identical, and each job J_i is associated a specific subset of machines $M_{J_i} \subseteq M$ where it can be processed (i.e. different jobs can have different job-machine mapping). The number of jobs which can be processed feasibly is reduced compared to the case with identical machines. The authors show the new problem is NP-complete by using a reduction from 3-SAT, and it remains NP-complete even if all jobs have equal value and we are asked to determine whether all can be scheduled. The NP-completeness can also be proved considering that S_{FSE} with non-identical machines is again representable using interval graphs, with each interval now taking the color from a subset of m. In the literature, this problem is called *list-coloring* on *interval graphs*.

***List*-coloring on interval graphs** - The *list*-coloring problem generalizes the version m-coloring by allowing a specific set (or *list*) of available colors for each vertex. Given a graph G and a finite list $L(v) \subseteq \mathbb{N}$ for each vertex $v \in V$, the list-coloring problem ask for a list-coloring of G, i.e. a coloring f such that $f(v)$ $\in L(v)$ for every $v \in V$. The *list*-coloring problem is NP-complete for many types of graph, e.g. IG [20]. In particular, note that the precoloring extension problem is a special case of list-coloring - with each uncolored vertex allowing the full list of m colors, and each precolored vertex having a list of a single element, i.e. the preassigned value. The NP-completeness of the precoloring extension on IG then implies that list-coloring on IG is also NP-complete.

S_{FSE} with machines ordered by capacity

In S_{FSE} with non-identical machines, each job J_i could be associated to any generic subset of machines $M_{J_i} \subseteq M$. For example, considering a set of m = 5 machines M = $\{M_1$, M_2, .., $M_5\}$, two jobs J_a and J_b could have (respectively) a job-machine mapping M_{J_a} = $\{M_1$, M_4, $M_5\}$, and M_{J_b} = $\{M_2$, M_3, M_4, $M_5\}$. Thus, jobs could have a subset of machines containing *holes*. In the example, M_{J_a} has a hole between M_1 and M_4. Further, any two jobs J_a and J_b could have job-machine mappings such that $(M_{J_a} \setminus M_{J_b} \neq \emptyset) \wedge (M_{J_b} \setminus M_{J_a} \neq \emptyset)$, i.e. one job could be processed into some machines which cannot be used for the other job and vice versa. For instance, in the example, M_1 appears only in M_{J_a}, while M_2 and M_3 are only in M_{J_b}.

Machines are often categorized by capacities, so that if a job can be processed by a machine of a certain capacity then any other machine with equal or greater capacity can also process the job. Let the set of machines M = $\{M_1, M_2, .., M_m\}$ be $ordered$ by decreasing capacity. Note that, given m machines, we can always find a permutation $\{h_1, h_2, ..., h_m\}$ of $\{1, 2, .., m\}$ such that $\{M_{h_1}, M_{h_2}, ..., M_{h_m}\}$ is ordered by capacity. Then, S_{FSE} *with machines ordered by capacity* is a particular case of S_{FSE} with non-identical machines, where any job-machine mapping M_{J_a} is such that $M_i \in M_{J_a} \iff M_j \in M_{J_a}, \forall\, j < i$. Now, job-machine mappings allow no holes, and further, any two jobs J_a and J_b have mappings such that $(M_{J_a} \subseteq M_{J_b}) \vee (M_{J_b} \subseteq M_{J_a})$. This means that, either both jobs can be processed by the same set of machines, or one of the jobs has a set of machines which properly contains the set of machines of the other job, i.e. jobs are *ranked*. The new problem of finding a feasible schedule under ordered machines can be modelled as *μ-coloring* on *interval graphs*.

μ-coloring on interval graphs - The μ-coloring problem [19] is a particular case of list-coloring. Given a graph G and a function $\mu : V \to \mathbb{N}$, G is μ-colorable if there exists a coloring f of G such that $f(v) \leq \mu(v)$ for every $v \in V$. In [20] it is shown that μ-coloring on interval graphs is NP-complete. The proof is based on the NP-completeness of the coloring problem on circular-arc graphs [43].

To represent the scheduling problem as μ-coloring on IG, each job J_i is attached a value $\mu(J_i)$ = $\max_j\, \{M_j \in M_{J_i}\}$. Note that $\mu(J_i)$ is sufficient to represent the set of allowed machines for J_i, i.e.:

$$M_{J_i} = \{M_j : j \leq \mu(J_i)\}$$

The interval graph is then μ-colorable if there exists a coloring of all intervals (or jobs) such that $f(J_i) \leq M_{\mu(J_i)}, \forall\, i = 1, .., n$. As μ-coloring on IGs is NP-complete, so is S_{FSE} with ordered machines.

S_{FSE} with machines ranked by job size

We saw before how μ-coloring models S_{FSE} with machines ordered by capacities, so that if a job can be processed by a machine of a certain capacity then any other machine with equal or greater capacity can also process the job. In real applications, the value of a job typically reflects an order size (e.g. number of pieces involved in the production of the order). By default, small jobs may not be allowed to execute on high capacity machines, as the cost of running a big machine may not be profitable on a small job, and further, this could waste the potential to execute a future order of greater size and value. Then, we give each job J_i a range (without holes) of machines from the set M=$\{M_1$, .., $M_m\}$ (ordered by decreasing capacity) where the job is allowed to execute. Jobs of same size are given the same range, and for increasing sizes the range shifts towards higher capacities. The new problem can be modelled using (γ,μ)-*coloring* on *interval graphs*.

(γ,μ)-coloring on interval graphs - (γ,μ)-coloring [20] is a generalized case of μ-coloring. Given a graph G and functions $\gamma,\ \mu : V \rightarrow \mathbb{N}$ such that $\gamma(v) \leq \mu(v)$ for every $v \in V$, G is (γ,μ)-colorable if there exists a coloring f of G such that $\gamma(v) \leq f(v) \leq \mu(v)$ for every $v \in V$. As μ-coloring is NP-complete, so is (γ,μ)-coloring.

To represent the scheduling problem as (γ,μ)-coloring on IG, each job J_i is attached values $\gamma(J_i)$ = $\min_j\ \{M_j \in M_{J_i}\}$ and $\mu(J_i)$ = $\max_j\ \{M_j \in M_{J_i}\}$. The values $\gamma(J_i)$ and $\mu(J_i)$ are sufficient to represent the set of allowed machines for J_i, i.e.:

$$M_{J_i} = \{M_j : \gamma(J_i) \leq j \leq \mu(J_i)\}.$$

Note that, for any two jobs J_a and J_b, the logic expression $(M_{J_a} \subseteq M_{J_b}) \vee (M_{J_b} \subseteq M_{J_a})$ is no longer true, i.e. one job could be processed by some machines which cannot be used for the other job and vice versa.

The interval graph is then (γ,μ)-colorable if there exists a coloring of all intervals (or jobs) such that $M_{\gamma(J_i)} \leq f(J_i) \leq M_{\mu(J_i)}$, $\forall\ i$ = 1, .., n. As (γ,μ)-coloring on IGs is NP-complete, so is S_{FSE} with machines ranked by job size.

Summary: hierarchy over S_{FSE} versions

In the previous pages we have introduced five versions of S_{FSE}:

1. with identical machines (S_{FSE-IM});
2. with identical machines and preassigned jobs $S_{FSE-IM-PJ}$;
3. with machines ordered by capacity (S_{FSE-OC});
4. with machines ranked by job size ($S_{FSE-RJS}$);
5. with non-identical machines ($S_{FSE-NIM}$).

We saw how the five versions can all be represented using interval graphs IG(V,E), and, in the order, can be solved in terms of:

1. m-coloring;
2. precoloring extension;
3. μ-coloring;
4. (γ,μ)-coloring;
5. list-coloring.

The problem of m-coloring on IG is a special case of precoloring extension with no preassigned vertex, and of μ-coloring with $\mu(v) = m$ for every vertex v in V. The problem of precoloring extension on IG is a special case of (γ, μ)-coloring, with $\gamma(w) = \mu(w)$ for every preassigned vertex $w \in W \subseteq V$, and with $\gamma(v) = 1$ and $\mu(v) = m$ for every other vertex $v \in V \setminus W$. The problem of μ-coloring on IG is a special case of (γ, μ)-coloring with $\gamma(v) = 1$ for every vertex v in V. Finally, the problem of (γ, μ)-coloring on IG is a special case of *list*-coloring with $\gamma(v)..\mu(v)$ not allowed to contain holes for every vertex v in V.

In conclusion [20], coloring (on IG) and correspondent scheduling problems of type (1) are in P, while coloring (on IG) and correspondent scheduling problems of type (2), (3), (4), and (5), are in NP-complete.

3.2.4 Scheduling for restaurant table allocation

The problem of restaurant table allocation (RTA), which is at the center of this dissertation, has been modelled as *scheduling jobs with fixed start and end times* (S_{FSE}), where jobs represent parties (or dinners) and resources represent tables. The 5 versions of S_{FSE} discussed in Section 3.2.3 take the following meaning within the problem domain of RTA:

1. S_{FSE-IM} models restaurants with all identical tables, e.g. 20 tables all of capacity 4. Perhaps not many restaurants allows such a simple model.

2. $S_{FSE-IM-PJ}$ models restaurants with all identical tables, and such that some parties have preassigned (or fixed) allocation. During floor management, for example, a schedule (or seating plan) must be reviewed after a dinner lasting longer than expected clashes with the next dinner planned on the same table, however parties who are already dining have a fixed table and so cannot be reallocated.

3. S_{FSE-OC} models restaurants with tables of different capacity, e.g. 20 tables, 8 two-seater, 6 four-seater, 4 six-seater, and 2 eight-seater. Further, there is *full-nesting* [13], i.e. any party of size s can go into any table of capacity s or more. In the example, a party of 2 can use any table of capacity 2 to 8.

4. $S_{FSE-RJS}$ models restaurants with tables of different capacities. However, there is *partial-nesting* [13], i.e. parties of size s can only go into tables of capacity in the range $s..s+\delta(s)$, where $\delta(s)$ is the maximum number of table seats which are allowed to remain unused - to guarantee a certain level of table occupancy. For example, a party of 2 might be restricted to use any table of capacity 2 to 3, even though the restaurant has larger tables.

5. $S_{FSE-NIM}$ models restaurants with generic non-identical tables. Each party will still have a set of possible tables (depending on the allowed level of nesting), but the set can now contain holes, e.g. to represent preferences. For example, a restaurant may have 3 four-seater tables $\{T_1, T_2, T_3\}$, 2 five-seater $\{T_4, T_5\}$, and 1 six-seater $\{T_6\}$, where T_1, T_4, and T_6 are special tables (e.g. booths). The restaurant may need to allocate a very important

party (VIP) of size 4 with a preference for a booth. The case can be tackled considering the VIP party having the set of possible tables with holes $\{T_1, T_4, T_6\}$.

The basic model designed in this dissertation is based on S_{FSE-OC}, i.e. the model with tables ordered by capacity, supporting full-nesting, and representable using μ-coloring on interval graphs (details will be presented in Chapter 4).

3.3 Restaurant revenue management

Revenue management is at the core of any business. The general goal is to choose the best configuration of resources, and to make the best use of them, ensuring that the margin is maximized so that the income is greater.

The growth of information systems has boosted the business of many companies in the last decades. Large size firms in sectors such as manufacturing, retail, transport, telecommunication, or financial, can now develop decision strategies (e.g. for resource planning, resource allocation, and resource pricing), based on complex optimization models. Sectors involving businesses of smaller size, like the restaurant industry, have not been as ready to invest into the use of informatics. Research in restaurant revenue management has stepped up only in recent years, but its application remains quite sporadic.

3.3.1 State of the art

Revenue management (RM), as reported in [62], is the “application of information systems and pricing strategies to allocate the right capacity to the right customer at the right place at the right time”. The authors analyze several issues which must be taken into consideration in order to achieve effectiveness for the specific case of *restaurant revenue management* (RRM).

Traditionally, the goal of many restaurateurs has been to maximize *seat occupancy*, aiming to use the full capacity of each table and to minimize the time between meals. However, seat occupancy can be influenced by other factors like kitchen size, menu items, and staffing level. The kitchen size may limit the number of meals that can be prepared at the same time. The preparation and consumption of different menu items can take different times, so even menu design can affect the total expected number of meals. Table usage is also dependent on the service level, i.e. the quantity and skills of staff operating over time. For example, a shortage of staff may delay the flow of dinners over the dining session, and therefore reduce the final turnover.

Even with the maximum seat occupancy, the revenue may not be optimal. For example, it may be better to have 60% occupancy and an average cheque per person of EUR 35 than a 100% occupancy with an average of EUR 15 per person. The *average cheque* has been the most important factor for some restaurateurs, but again, an evaluation based on this measure is still approximate. In a cheque of EUR 15 there may be a *margin* of profit of 10, while in a cheque of EUR 35 the margin may be 15, e.g. because the food is more expensive, or more difficult to prepare, thus requiring an expensive master chef. Similarly, cheque (and margin) must be related to the *duration* of the meal. A party of two spending EUR 50 with a margin of 20, but occupying the table for 3 hours, is perhaps equivalent to two parties of two spending EUR 25 with a margin of 10 each, and occupying the table for 1.30 hours.

Kimes *et al* [62] model restaurant tables over time as *perishable inventory* - if some table seats are not being used for some periods, that part of inventory perishes. The authors introduce a time-based revenue performance measure, named *revenue per available seat hour* (RevPASH). RevPASH expresses a better estimate of the different aspects (discussed above) that contribute to restaurant efficiency.

Given a time period T over a dinner session, RevPASH can be defined as follow:

$$RevPASH(T) = \frac{Revenue(T)}{Number\ of\ Available\ Seats\ (T) \times T},$$

where *Revenue(T)* is a function of price of the menu items sold during T, and of ingredients, kitchen, and service costs to make and provide the items.

Maximizing RevPASH has been identified as the real goal for restaurateurs. The optimization of RevPASH can be split into two main branches: *demand based pricing* (or *menu engineering*) and *duration management*.

Pricing management

Traditional pricing policies are cost based, i.e. menu items are priced in order to achieve a certain margin, which is defined as price minus cost of the ingredients. In *demand based pricing* [62], the first objective is to improve seat occupancy during off-peak hours by offering discounts or specials. This allows good value meals for the customer who has time flexibility, or who is reluctant to pay the full price, while those who are not price sensitive can pay more and buy their dinner at peak time. A combination of *cost* and *demand* based pricing is represented by *menu engineering* (ME). In ME, menu items are divided into four classes, depending on margin (m, or price minus cost) and demand (d, or volume of items sold), i.e.: (1) m high and d high; (2) m low and d high; (3) m high and d low; (4) m low and d low. In the first two cases the demand is high so the restaurant can try to increase the price. In the third case the margin is high but the demand is low, so the restaurant can try to reduce the price.

Duration management

In *duration management* [62] the main issues concern reducing the uncertainty on customer *arrival time* and on *meal duration*, and the *time between meals*.

Arrival times can be forecasted with some approximation based on experience or using historical data (if available). Otherwise, some restaurants reduce uncertainty by selling only fixed times, e.g. 12 p.m., 2 p.m., 4 p.m., .., 10 p.m., but this requires making sure each dinner is over in 2 hours.

Restaurants taking reservations have a better knowledge on the set of parties which are going to arrive, but the drawback is that reserved tables have to be kept unavailable until the party arrives, and often, parties are late or do not show up at all. Restaurants not taking reservations manage arrivals in queues. The knowledge on how parties are distributed by number, time, and size, is limited to the current queue. The advantage is that the manager knows exactly who is available to start, so tables are never kept idle when a party is waiting at the door.

Uncertainty in arrivals can be better managed if the table mix is optimized for the standard party mix of the restaurant. Consider two table mixes for the same restaurant floor, both allowing the same number of seats, one with more but smaller tables, and the other with fewer but larger tables. We assume tables cannot be joined with each other. If the demand is mainly composed of small parties, the table mix with smaller tables will be able to accommodate more people than the table mix with larger tables. Vice versa, if the demand is mainly of large parties, the mix with larger tables can accommodate more people than the mix with smaller tables.

Even the waiting time of parties in the queue, and therefore the time between meals, would be reduced with a better table mix. Consider a first example where the expected party mix is mainly composed of twos. Consider a 4-seater table currently occupied by a couple, and another party of two waiting for that same table to become available. This second party could have been served immediately if the table mix included a pair of 2-seater tables rather than that 4-seater. Analogously, consider the opposite case with party mixes of larger average size. Consider a pair of 2-seater tables currently free, and a party of 4 waiting for a 4-seater to become available. This party could have been accommodated immediately if the table mix included a 4-seater rather than the two 2-seater tables.

The duration of a dinner is a function of customer behavior but also of restaurant behavior. It is important for the restaurant to avoid low profit time on tables. Making the dinner cycle more time effective can be achieved through improving menu design, service procedures, staffing level, and communication between kitchen and floor. Menu items which take too long in preparation or consumption should not be included in the menu. Service procedures can be improved by tightening operations like greeting, seating, taking and delivering orders, table bussing, and bill delivery. Speeding up service procedures may require staff training, and also increasing the number of staff. The cost of paying the training and the salary for the extra labor has to be balanced with the revenue coming from the increase in meals being served. Finally, some restaurants have already introduced table management systems based on hand-sets which allow instant communication between kitchen and floor staff - e.g. saving the physical time required by waiters to walk to the kitchen to notify new orders.

Implementing restaurant revenue management

In [61] a five-step procedure has been suggested for implementing revenue management in restaurants. The approach involves: (1) establishing the baseline, i.e. retrieving data distributions about average cheque, seat occupancy, party arrivals, meal times, RevPASH, and party mix; (2) understanding the drivers, i.e. analyzing the causes affecting meal times and RevPASH; (3) developing a strategy, e.g. improving meal durations, table mix, menu design, service efficiency, or customer arrival management; (4) implementing the changes, e.g. investing in training or technology, or reviewing menu, table mix, or staffing level; (5) monitoring outcomes, i.e. evaluating the changes in terms of revenue, operation, and customer satisfaction.

In [62] the 5-step revenue management strategy is implemented in a real restaurant with 230 seats and with no reservations. The applied strategy purely involves *duration management*, while the complementary approach concerning *demand based pricing* (as discussed in the previous pages) was not considered, i.e. the original menu and prices were preserved. They show how the restaurant revenue was increased by focusing on increasing seat occupancy, and on reducing meal duration (in mean and variance). In particular, the revenue management strategy consisted in introducing a table mix (based on [90]) more suitable for the standard party mix of the restaurant, and in making the service process more efficient, especially in payments and table bussing. The reduction of meal duration (mean and deviation) was tackled by reviewing the hosting procedures, investing in focused training to staff, buying a table management system, buzzers to notify when tables become ready, and faster credit card processing tools, and hiring extra personnel where required. The costs of implementation were shown to be covered in one year, and the increase in yearly profit was estimated to be 5%.

Optimal restaurant table configuration

Above we have discussed the importance of using a table mix suitable for the standard party mix of the restaurant. However, the party mix can be unknown, e.g. for a brand new restaurant, or can be quite variable depending on the day, or on the time of the day. Uncertainty in party mix can be managed with more flexibility if we have a set of combinable tables, as the combinability allows more options in serving different mixes of parties. In particular, when party mixes vary over time, combinability allows multiple restaurant configurations, which can be changed by day, or even during the same dining session. The gain in flexibility has to be balanced with the drawback that any group of combinable tables rarely become available all at the same time, so those which are freed first have to be hold for as long as all the other tables in the group remain unavailable.

In [90] simulation is used to determine the best restaurant configuration, i.e. the best mix of combinable and non-combinable (or dedicated) tables. Results show that combinable tables have a benefit for small restaurants (represented with 50 seats), while for large restaurants (represented with 200 seats) it is better a table mix purely composed by dedicated tables. Results also suggest how revenue could be increased by changing the restaurant configuration day by day, according to the expected party mix - provided this will not annoy the customer or create confusion for the staff.

The simulation model in [90] considers a restaurant with no reservations, where tables are distributed over 4 different capacities (i.e. 2, 4, 6, 8), parties arrive and get seated according to the rule that any table becoming free is assigned to the largest waiting party - thus penalizing small parties (especially singletons) - and tables can be combined only to serve parties of size 5 to 8 (8 is the maximum party size). This brings to a total of some 8000 different table mixes. As pointed out by the author, different settings - e.g. more table sizes, a seating rule based on the fairer and common first-come-first-serve principle (FCFS), or a different rule to decide which size of party can use combinable tables - may change the problem size and the results.

The main issue for trying different settings is the fact that for restaurants of larger size (i.e. more than 200 seats), or with more than four table sizes, the enumeration of all possible table mixes becomes intractable. The restaurant we are going to model in this thesis has about 100 seats, but tables are required to cover sizes from 2 to 30 (according to the standard party mix), which makes a complete table mix optimization perhaps impractical. In [90], a faster optimization algorithm is one of the issues the author left for future analysis.

Open issue in booking and seating optimization

Restaurants can manage customer arrivals through methods like overbooking or forecasting. In traditional systems, the restaurant manager handles booking and seating decisions “simply” based on his experience. The use of on-line computerized reservation and table management systems has recently started to grow also for the restaurant business [73] [60]. However, current solutions do not support any advanced form of booking or seating optimization. For instance, quoting [62], in current on-line systems restaurant managers “must decide on the number of tables to allocate to each time slot and determine the interval between reservations”, and “little research exists on the optimal number of tables to allocate to each time slot”. The research discussed in this dissertation aims to cover this gap, i.e. to show how artificial intelligence can be used to support on-line optimization, in both booking and floor management.

Optimization based models

To our knowledge, the most relevant work on developing optimization models for restaurant table management is presented in [13]. The general problem is to decide whether to accept or reject parties as they arrive, and which table to allocate to each accepted party, maximizing the revenue (represented by the number of people eating over a dining period), while controlling the average time parties have to wait before getting seated, as well as the degree of fairness violation (i.e. fairness being represented by the FCFS principle). The model assumes that every party can be seated at any time within a maximum waiting period - which is fixed to 1 hour for parties with reservation, and to half an hour for walk-ins - but parties are allowed to leave the queue only at the end (i.e. not earlier nor later) of that period. The authors start by presenting optimization models which do not consider reservations, and then incorporate reservations.

For the case with no reservations, tests have been dimensioned over a real restaurant of small size, with 9 tables and a total of 38 seats. Customer arrivals have been modelled as a non-homogeneous Poisson process - with rate λ(arrival time, party size) - as the real distribution was unknown. They compare three different levels of *nesting*, i.e. allowing parties of size s to use only tables of size $s'=s$

(no-nesting), $s \leq s' \leq s+1$ (1-up-nesting), or $s' \geq s$ (full-nesting). Using nesting, the issue is on whether to assign a party to an oversized table, or to reject the party and wait for one of more suitable size but whose arrival is uncertain. Simulation results indicate that nesting is good for small restaurant loads - the version using no-nesting saves the large tables for large parties, but this does not payoff due to the low number of arrivals. For larger loads the benefit of nesting decreases, as there is a much higher probability of a party of the right size arriving (later), and so we lose if we put a small party at the table. Overall, the optimization models are shown to outperform other comparison methods (e.g. FCFS) for low, medium, and particularly for high restaurant loads. The best optimization method, which is based on approximated dynamic programming (ADP), achieves a higher revenue than FCFS, serving about the same number of parties (but a more profitable set), and maintaining (in some cases decreasing) the average waiting time.

For the case with reservations, tests have been carried out using real data from a real restaurant with 27 tables and 86 seats, and assuming no-nesting (for simplicity). Table demand was generated using the real distribution, and tests were repeated over a lower (90 people) and higher (120 people) restaurant load. For both loads the performance achieved using optimization was 3.5 % to 8.9 % better than FCFS in terms of revenue, with comparable waiting time.

In all the experiments, without or with reservations, the optimization models were set with constraints to guarantee the FCFS fairness rule over parties with same size. However, this still allows unfair situations with parties of larger size being served before smaller parties which are in front of them in the queue. For example, a party of 4 may get quite annoyed seeing a party of 6 behind them being called first. Further, in the experiments tackling reservations no nesting has been allowed. This is quite restrictive, e.g. preventing parties of 4 from using tables for 8 can be a possible policy, however tables for 6 (at least) are generally allowed to seat fours. Results should be evaluated over a (more standard) seating policy based on some level of nesting. Finally, as pointed out by the authors, the optimization models have so far represented restaurants with non combinable (or dedicated) tables, and the extension to consider combinability is stated as future research.

3.4 Discussion

In the first part of this chapter, we gave an introduction to constraint programming and the standard solving procedure (tree search interleaved with constraint propagation), and presented the relevant techniques available from the literature to solve dynamic problems with uncertainty. In the next paragraphs, we will briefly revise the reasons why we have not chosen the existing approaches to tackle Restaurant Table Management (RTM).

Probabilistic [36] and Stochastic [67] CSPs are suitable for problems which have an accurate probability model for changes, which is unrealistic for RTM. Further, computing a model and maintaining it up to date in such a dynamic environment might be too expensive to do effectively.

Branching CSPs [38] model problems whose future development can be represented using probabilistic trees, where one event (node) depends on other events (nodes) earlier in the tree. In RTM most events appear to be independent.

Sampling methods [10] [11] [12] [26] could be applied to tackle RTM, e.g. to decide, at each point during a dinner session, the seating plan that maximizes the chances to accommodate future meal requests. However, the computation of samples of sufficient size and in sufficient number to effectively optimize RTM might again be too complex to do it in practical time.

In RTM, frequent table reallocations and reconfigurations cause confusion and can annoy both staff and customers. The approach based on Local Changes [91] is not suitable for RTM, e.g. to accommodate a delay, Local Changes could return a solution which requires the reallocation of a considerable number of tables, even though there was another possible solution with only a few changes. In our study on solution stability for RTM, we will develop a search algorithm for Optimally Stable Solutions, similar to [79].

We also argued against the usability of Slack Times [30] to protect schedules from disruptions and delays. An accurate and effective choice of slack times is not easy to obtain, considering that disruptions and delays are caused by highly uncertain factors like customer behavior and restaurant performance. Restaurant profit and table usage may deteriorate if slack times are not positioned in the correct place and in the correct quantity.

Just In Case scheduling [35] pre-computes contingent schedules to anticipate likely changes and breakages. In RTM, the possible combinations of changes and breakages that can occur over an evening session are huge, so implementing contingent schedules would not be of practical use.

The fact that changes are very frequent and diverse in RTM makes the implementation of Super Solutions (SS) [55] complex to do effectively. Further, SS may not be suitable to model problems which requires us to accommodate chains of changes (like RTM).

In the second part of this chapter, we described the problem of scheduling, focusing on scheduling jobs with fixed start and end time (S_{FSE}), and providing representations for restaurant table allocation. We finally discussed restaurant revenue management in general, including pricing, menu, and meal duration management, a work on table mix optimization, and an optimization model for handling queues in table management.

The next chapter begins the central part of the dissertation, and concerns the development and testing of our initial (constraint) model of the problem. The problem will be represented as a version of S_{FSE}.

Chapter 4

Modelling and solving the static decision problem

4.1 Introduction

In this chapter we discuss our basic model for tackling the restaurant problem. We present a solution that deals with booking and floor management together. Specifically, we consider table management as a sequence of static decision problems linked by changes. Each time a change happens, e.g. a new request, a booking arriving early or late, or a dinner lasting shorter or longer than expected, we reformulate and solve a new static problem.

In the next section we represent table management as a scheduling problem, with tables as resources and parties as tasks with fixed start and end times. We then introduce an initial (basic) CSP representation of the static problem. We extend the original representation with some pre-solving checks and additional constraints designed for improving the efficiency through enforcing extra constraint propagation. We present a study concerning the design of search algorithms, again, finalized to produce a more efficient search. We conclude by showing the benefit achieved on both representation and search algorithms from comparing the solution we finally achieved to the original, and to others recommended by the literature. We test our model on the original scheduling problem with 23 tables (i.e. as in the Eco restaurant), we then scale to a bigger problem size (i.e. on an hypothetical restaurant of 100 tables), and we finally generalize our search algorithm and apply it to a different problem class (i.e. on quasi-groups of order 20).

The aim of this chapter, with respect to our thesis, is to show how constraint programming can be used to model and solve the static decision problem, and how careful modelling can improve the efficiency of solving. Our current interest is to demonstrate how our model can provide answers in reasonable time to be usable in a restaurant, while in the next chapter we will investigate whether we can also give good quality solutions (or table plans). In table management, either booking or floor phase, decisions have to be taken in real time. As a general guide, we expect a manager to take an average of 10 to 20 seconds to decide on where to allocate a new booking, or on how to reallocate a seating plan after a change.

4.2 The scheduling problem

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 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. Figure 4.1 (top) shows a problem instance with four parties (left) and a possible allocation (right). Note that P_2 is allocated into T_2 and T_3, since the problem allows T_3 to be joined onto T_2 to give a capacity of 6.

Any change generates a new instance, e.g. with an extra element in the set of parties, if the change regards a booking request, or with a different start or end time for one of the original elements, in case of a late (or early) arrival (or finish). Figure 4.1 (left) represents a chain of 3 static instances linked by 2 changes (i.e. a new booking and a late finish). On the right we see possible seating plans for the 3 instances - in the second and third case, the changes in table occupancy from the previous plan are highlighted in gray. More specifically, on the top we see the initial problem at time 0, with 4 parties (left) and a possible allocation (right). In the middle we assume we are at time 1, i.e. P_1 is already occupying table T_1, P_2

is on tables T_2 and T_3, and there is a new booking request (P_5). Again, we have the second instance with the extra party (left), and a possible allocation (right). Finally, we assume we are at time 2, i.e. P_2 has just freed T_2 and T_3, P_3 is on T_4, however P_1 is not ready to free T_1 as expected. At the bottom we see the new problem, and a possible reallocation.

Party	**Size**	**Start**	**End**	**Table**
P_1	2	0	2	?
P_2	4	0	2	?
P_3	3	1	3	?
P_4	2	2	4	?

Table[size]	**0**	**1**	**2**	**3**
$T_1[2]$	P_1	P_1	P_4	P_4
$T_2[3]$	P_2	P_2		
$T_3[3]$	P_2	P_2		
$T_4[4]$		P_3	P_3	

P_1	2	0	3	T_1
P_2	4	0	2	$T_{2,3}$
P_3	3	1	3	?
P_4	2	2	4	?
P_5	2	2	4	?

Table[size]	**0**	**1**	**2**	**3**
$T_1[2]$	P_1	P_1	P_4	P_4
$T_2[3]$	P_2	P_2	P_5	P_5
$T_3[3]$	P_2	P_2		
$T_4[4]$		P_3	P_3	

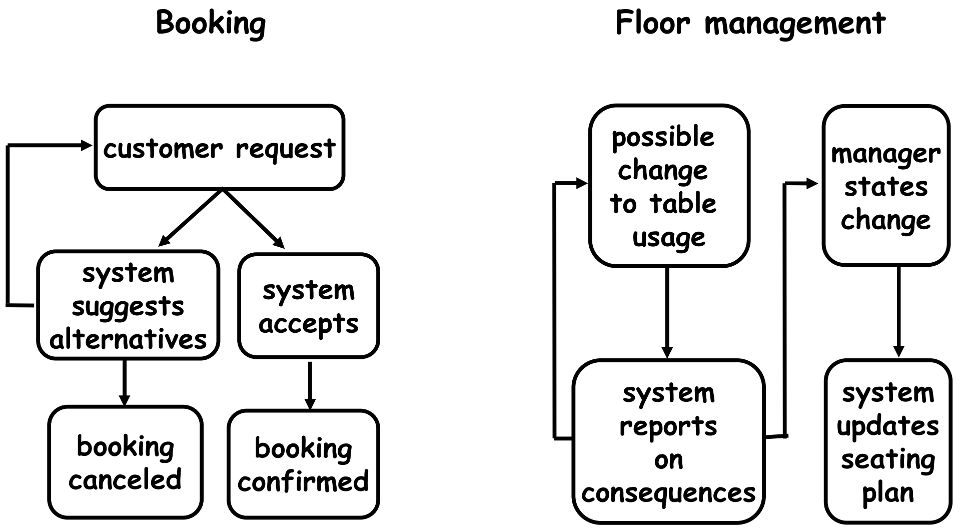
P_1	2	0	3	T_1
P_2	4	0	2	$T_{2,3}$
P_3	3	1	3	T_4
P_4	2	2	4	?
P_5	2	2	4	?

Table[size]	**0**	**1**	**2**	**3**
$T_1[2]$	P_1	P_1	P_1	
$T_2[3]$	P_2	P_2	P_5	P_5
$T_3[3]$	P_2	P_2	P_4	P_4
$T_4[4]$		P_3	P_3	

Figure 4.1: Sequence of changes: time 0 (top), initial booking sheet with 4 parties and possible seating plan; time 1 (middle), new booking request P_5 for time 2, and possible allocation into the current plan; time 2 (bottom), late finish of party P_1, and possible seating reallocation.

4.2.1 Managing infeasibility

If there is no way to accommodate a change by rearranging the seating plan, reallocating everybody, but maintaining their original time slots (i.e. start time and duration), the problem is found to be infeasible, and the static model simply reports that there is no schedule which can seat the new set of parties. In the real application, depending on the type of change, further steps may be considered or even necessary. Figure 4.2 reports two use cases which explain how our solution could be utilized in the real application, to provide assistance to the restaurant in managing feasible and infeasible changes. Typically, left, when infeasibility is caused by a new booking request, the restaurant can go further and suggest possible alternatives to the customer. Another case, right, is when infeasibility is caused by a change in table usage - for example, such change can be a party arriving late or a dinner that is longer than expected. This forces the restaurant to rearrange the seating plan, e.g. delaying some of the bookings which have not yet been seated. Chapter 6 discusses more details about what we can do in these situations in our application.



Booking
customer request
system suggests alternatives
system accepts
booking canceled
booking confirmed
Floor management
possible change to table usage
system reports on consequences
manager states change
system updates seating plan

Figure 4.2: Use cases: (left) booking phase; (right) floor phase.

4.3 Basic CSP representation

In this section we introduce a basic constraint representation of the problem.

4.3.1 Scheduling using single tables

Our first model does not consider table configurations, i.e. parties can only be accommodated into single tables. Figure 4.3 displays an example. With single tables, and despite having fixed start and end times, our scheduling problem is NP-complete [20] (discussed in Section 3.2). Note that, the problem can also be regarded as the "Reservations without Slack" problem (feasibility version) [76].

Party	**Size**	**Start**	**End**	**Table**
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**
$T_1[2]$	P_1	P_1	P_4	P_4
$T_2[3]$		P_3	P_3	
$T_3[3]$			P_5	P_5
$T_4[4]$	P_2	P_2		

Figure 4.3: Problem instance (top); and a possible seating plan with no table configuration (bottom).

We want to ensure we can provide an effective solution to this simplified problem. In the next chapter the model will be extended to also represent multiple joining of tables. Note that, as from our discussion in Chapter 2, by allowing different and dynamic restaurant layouts the number of possible allocations increases and the search space explodes with the number of possible configurations.

Modelling using μ-coloring on interval graphs

The basic model designed in this thesis is based on S_{FSE-OC} (Section 3.2), i.e. scheduling with fixed start and end times and with tables ordered by capacity. S_{FSE-OC} is representable using μ-coloring on interval graphs. For clarity, in the next example we discuss the μ-coloring problem representation of the simple scheduling instance of Figure 4.3.

Example - Consider a set of 4 tables $\{T_1, T_2, T_3, T_4\}$ of capacity $\{2, 3, 3, 4\}$, and a set of 5 parties $\{P_1, P_2, P_3, P_4, P_5\}$ of size $\{2, 4, 3, 2, 2\}$ (as shown in Figure 4.3). Figure 4.4 (top) shows the 5 parties with fixed start and end times represented as a set of intervals over the real time-line. For each party, there is a set of tables that can accommodate the party. In particular, if a party can be accommodated by table T_i then it can also be accommodated by any other table of capacity at least T_i.capacity, i.e. by any other T_j with $j > i$. Figure 4.4 (middle) represents the correspondent problem instance of μ-coloring on interval graphs (we use the notation $T_i = i$). The graph has 5 nodes represented by parties P_1, P_2, P_3, P_4, P_5. If two intervals intersect each other over the real line then the correspondent nodes are connected (and must be assigned different values). There are two cliques, which correspond to the intersections at times t_1 and t_2 in Figure 4.4 (top). We use a μ function $\mu(P_i) = \min_j\{T_j \text{ can accommodate } P_i\}$, i.e. $(\mu(P_1), \mu(P_2), \mu(P_3), \mu(P_4), \mu(P_5)) = (1, 4, 2, 1, 1)$. Note that $\mu(P_i)$ is sufficient to represent the set of allowed tables for P_i, which is $\{T_j : j \geq \mu(P_i)\}$. The coloring function $f : P_i \rightarrow \mathbb{N}$ maps each node to a color (represented by a number). The problem is μ-colorable if there exists a coloring f such that $f(P_i) \geq \mu(P_i)\ \forall\ i \in \{1, 2, 3, 4, 5\}$. Figure 4.4 (bottom) shows a possible solution.

4.3.2 CSP representation

To represent the scheduling problem as a CSP, we model the parties as decision variables, and the tables as the values to be assigned.

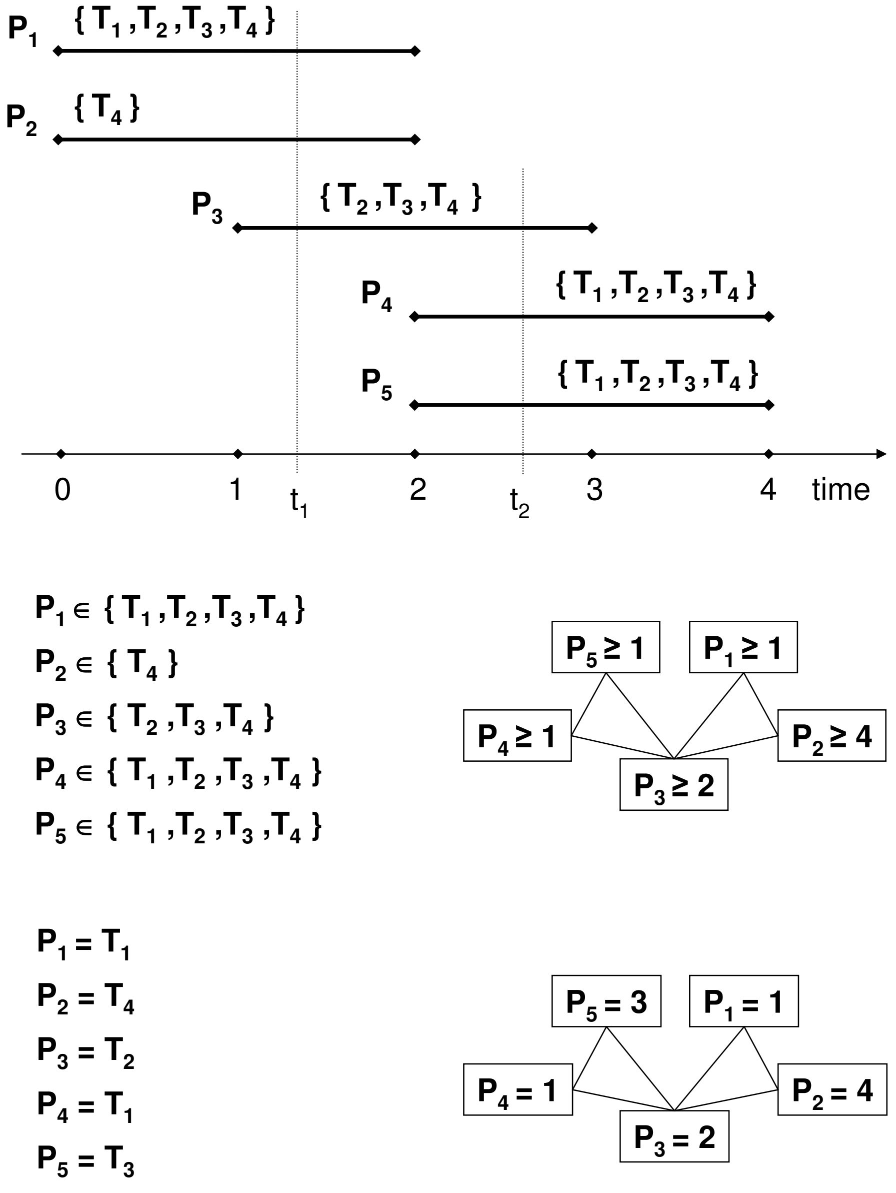


Figure 4.4: An example of μ-coloring on interval graphs representing an instance of S_{FSE-OC} for restaurant table allocation: set of intervals over the real line (top); problem instance (middle); possible solution (bottom).

Decision variables

We assume a set (or booking sheet) of N parties, $P = \{P_1, P_2, .., P_N\}$. Each party P_i is represented by a quadruple $P_i = \langle P_i.size, P_i.start, P_i.end, P_i.table\rangle$, where P_i.size is the size of the party, P_i.start is the dinner start time, P_i.end is the expected end time, and P_i.table is the table to be assigned, which also represents our decision variable. Therefore, our set of variables is:

$$X \;=\; \{\ P_1.table, P_2.table, .., P_N.table\ \}\ .$$

Possible values

We assume a set (or restaurant) of M tables $T = \{T_1, T_2, .., T_M\}$, where each table T_i has a capacity T_i.capacity. Our set of possible values is:

$$Y \;=\; \{\ T_1, T_2, .., T_M\ \}\ .$$

Initial domains

Each decision variable P_i.table can take values in a subset of Y, i.e. its initial domain is defined as $D_i = \{T_j \mid T_j.capacity >= P_i.size\}$. The set of initial domains is:

$$D \;=\; \{\ D_1, D_2, .., D_N\ \}\ .$$

Constraints

Our basic model is made by the two types of constraints: a *no-overlapping* constraint (C_{1_t}), which ensures that all parties overlapping in time use different tables; and a *capacity constraint on adjacent tables* ($C_{2_{khij}}$), which represents those cases where two adjacent tables cannot be fully occupied at the same time, e.g. if the space is not enough to fit all the chairs. We now explain C_{1_t} and $C_{2_{khij}}$ in detail.

Lets assume $R = \{0, 1, .., T\}$ be the range of possible start times. For each time t in R, we consider the set of parties overlapping in t, $X_t = \{P_i.table \mid P_i.start \leq t < P_i.end\}$. Then we can formally express:

$$\forall t \in R, \quad C_{1_t} = \textit{alldifferent}\{X_t\} \ .$$

Note that, there can be two distinct times $\{t_1, t_2\} \in R$, for which X_{t_1} contains or is equal to X_{t_2}. For our overlapping constraint, we need to consider only a subset $R^I \subseteq R$, such that there is no pair of times $\{t_1, t_2\} \in R^I$, $t_2 \neq t_1$, for which X_{t_1} contains or is equal to X_{t_2}. Formally, $R^I \subset R$ is defined so that:

$$\forall t_1 \in R^I, \ \neg\exists t_2 \in R^I, \ t_2 \neq t_1, \ s.t. \ (X_{t_1} \subseteq X_{t_2}) \vee (X_{t_2} \subseteq X_{t_1}) \ .$$

We can obtain R^I with the following step:

$$R^I = \{ \ t_1 \in R \mid \neg\exists \ t_2 \in R \setminus \{t_1\}, \ (X_{t_1} \subset X_{t_2}) \vee ((t_2 < t_1) \wedge (X_{t_2} = X_{t_1})) \ \} \ .$$

R^I contains only those $t_1 \in R$ for which X_{t_1} is not properly contained by any other X_{t_2}, with $t_2 \in R$, i.e. we only consider the times having dominant sets. Further, if any X_{t_2} is equal to any X_{t_1}, with $t_2 > t_1$, R^I excludes t_2, i.e. we include only the first time in case two (or more) have the same set. Figure 4.5 shows 5 parties, and the elements in X_t over time. The set of possible start times is $R = \{0, .., 7\}$, while, applying the condition above, the set of times relevant for the overlapping constraint would be $R^I = \{0, 4\}$. Note that the overlapping constraint would be redundant if applied on any other time.

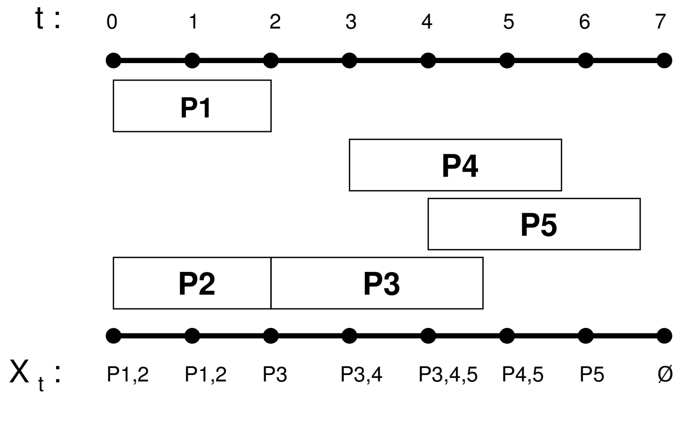


Figure 4.5: Example showing the set of overlapping parties over time.

Constraint C_{1_t} then becomes:

$$\forall t \in R^I, \quad C_{1_t} = \textit{alldifferent}\{X_t\}\ .$$

Let A be the set of pairs (k, h), such that T_k and T_h are adjacent tables, i.e.:

$$A = \{\ (k, h) \in (1..M, 1..M) \mid k\ \neq\ h, \textit{adjacent}(T_k, T_h)\ \}\ .$$

Let B be the set of quadruples (k, h, s_k, s_h), such that $(k, h) \in A$, and such that the pair of tables (T_k, T_h) cannot fit a pair of parties of sizes (s_k, s_h) at the same time, i.e.:

$$B = \{\ (k, h, s_k, s_h) \mid (k, h) \in A,\ s_i \leq T_i.capacity,\ \neg fit\ ((T_k, T_h), (s_k, s_h))\ \}\ .$$

Given B, the general procedure to define constraint $C_{2_{khij}}$ is then:

$$\begin{array}{lll} \forall & (k, h, i, j), & s.t. \\ & & (i, j)\ \in\ (1..N, 1..N), i\ < j\ , \\ & & (k, h, P_i.size, P_j.size)\ \in\ B\ , \\ & & \exists\, t\ \in\ R^I,\ s.t.\ (P_i.table \in X_t) \wedge (P_j.table \in X_t)\ , \\ & & \\ C_{2_{khij}} & = & (P_i.table \neq T_k) \vee (P_j.table \neq T_h)\ . \end{array}$$

The procedure considers any pair of adjacent tables (T_k, T_h) and any pair of parties (P_i, P_j) such that, the quadruple $(k, h, P_i.size, P_j.size)$ appears in the set B, and there exists a t in R^I where P_i and P_j overlap. $C_{2_{khij}}$ forbids the pair of assignments $(P_i.table = T_k)$ and $(P_j.table = T_h)$.

Notation

For convenience, from now on we may use P_i to refer to the decision variable P_i.table.

Final CSP

The set of variables, the set of possible values, the initial domains, and the constraints are summarized in Figure 4.6, in which we use the new notation.

Variables : $X = \{\ P_1, P_2, .., P_N\ \}$

Values : $Y = \{\ T_1, T_2, .., T_M\ \}$

Domains : $D = \{\ D_1, D_2, .., D_N\ \}$
$D_i = \{\ T_j \mid T_j.capacity >= P_i.size\ \}$

Constraints : $\forall\, t\ \in\ R^I\ ,$
$X_t = \{\ P_i \mid P_i.start \le t < P_i.end\ \}\ ,$
$C_{1_t}\ = \textit{alldifferent}\{X_t\}$

$A = \{(k,h) \in (1..M, 1..M) \mid k \neq h, \textit{adjacent}(T_k, T_h)\}$
$B = \{(k,h,s_k,s_h) \mid$
$\quad (k,h) \in A,\ s_i \le T_i.capacity,\ \neg fit\ ((T_k,T_h),(s_k,s_h))\}$

$\forall\ (k,h,i,j)\ ,$
$(i,j)\ \in\ (1..N, 1..N), i\ < j\ ,$
$(k,h,P_i.size,P_j.size)\ \in\ B\ ,$
$C_{2_{khij}}\ =\ (P_i \neq T_k) \vee (P_j \neq T_h)$

Figure 4.6: CSP representation: variables, values, domains, and constraints.

Example

Given a restaurant description (i.e. T, A, B) and a booking sheet (P), we have now all the information we need to define the corresponding CSP(X, D, C). Figure 4.7 shows the resulting constraint model for the simple problem of Figure 4.3.

Booking sheet and restaurant description - The booking sheet P is represented by Figure 4.3 (top). For the restaurant description (T, A, B), T can be retrieved in Figure 4.3 (bottom). In this example, we assume $A = \{(1,2),(2,3),(3,4)\}$ (i.e. tables with consecutive numbers are adjacent), and $B = \{(3,4,3,4)\}$ (i.e. tables T_3 and T_4 cannot simultaneously use their maximum capacity).

CSP - 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. Note that the time range is $R = \{0,1,2,3\}$, while the set of times used by our constraints is $R^I = \{1,2\}$. Constraints C_{1_1} (defined in $t = 1$) and C_{1_2} (defined in $t = 2$) ensure that all parties overlapping in time use different tables. C_{23432} (defined in $t = 1$) ensures T_3 and T_4 are not both fully occupied at the same time (which could only happen if they are assigned P_3 and P_2 respectively).

Variables :	P_1, P_2, P_3, P_4, P_5
Domains :	$D_1 = \{T_1, T_2, T_3, T_4\}$
	$D_2 = \{T_4\}$
	$D_3 = \{T_2, T_3, T_4\}$
	$D_4 = \{T_1, T_2, T_3, T_4\}$
	$D_5 = \{T_1, T_2, T_3, T_4\}$
Constraints :	$C_{1_1} = \mathit{alldifferent}(P_1, P_2, P_3)$
	$C_{1_2} = \mathit{alldifferent}(P_3, P_4, P_5)$
	$C_{23432} = (P_3 \neq T_3) \vee (P_2 \neq T_4)$

Figure 4.7: CSP representation for the example of Figure 4.3.

4.4 Additional checks and constraints

Using the basic CSP model presented above[†], in some initial tests some problem instances were taking too long to solve - some were still unsolved after several

[†]Coded in Ilog Solver 5.3

hours. This was not acceptable if we aim to have a practical and effective solution for table management. Therefore, as a first step to improve the efficiency, we introduced two types of pre-solving checks and one type of additional constraint for symmetry breaking. These are not necessary to solve the problem itself (C_{1_t} and $C_{2_{khij}}$ are sufficient for it), however they are designed to speed up the resolution process. Specifically, the pre-solving checks are introduced to spot infeasibility before starting the search process, while the additional constraints are aimed to prune the search space.

4.4.1 Pre-solving checks

Let RC be the maximum capacity of the restaurant, i.e.:

$$RC = \sum_{i=1..M} \{ \ T_i.capacity \ \} \ .$$

The number of covers seated at any time t must not exceed RC. Similarly, the number of parties seated at any time t must not exceed the number of tables M.

Let $maxNC$ be the maximum number of covers at any time t, i.e.:

$$maxNC = \max_{t \in R^I} \sum_{i=1..N, \ P_i.table \ \in \ X_t} P_i.size \ .$$

Let $maxNP$ be the maximum number of parties at any time t, i.e.:

$$maxNP = \max_{t \in R^I} \sum_{i=1..N, \ P_i.table \ \in \ X_t} 1 \ .$$

We can then define the following pre-solving checks:

$$C_3 \ = \ maxNC \leq RC \ ,$$

$$C_4 \ = \ maxNP \leq M \ .$$

Lets consider a small restaurant with 2 four-seater and one six-seater tables (i.e. $M = 3$). If the number of parties at a time t is 2, but the parties are both of size

6, then C_4 is not able to spot the infeasibility. C_4 only compares the maximum number of parties overlapping in time to the total number of tables. We extend C_4 to distinguish among categories of parties (by size lower bound) and of tables (by capacity lower bound). Back to our example, the new version will find that there are too many parties of size six (or more) compared to the number of tables which can serve six (or more).

Let $maxNP_s$ be the maximum number of parties of size at least s overlapping at any time, i.e.:

$$maxNP_s = \max_{t \in R} \sum_{i=1..N,\ P_i.size \geq s,\ P_i.table\ \in\ X_t} 1\ .$$

Let M_c be the number of tables of capacity at least c, i.e.:

$$M_c = \sum_{i=1..M,\ T_i.capacity \geq c} 1\ .$$

Let $minPS$ and $maxPS$ be the minimum and maximum party size, i.e.:

$$minPS = \min_{i=1..N} \{P_i.size\}\ , \qquad maxPS = \max_{i=1..N} \{P_i.size\}\ .$$

At any time t, the number of overlapping parties of size at least i must not exceed the number of tables of capacity at least i, for all possible $i \in \{minPS, .., maxPS\}$.

Therefore, we can formally express this as:

$$\forall\ i \in \{minPS..maxPS\}\ ,\quad C_{4_i}\ \ =\ \ maxNP_i \leq M_i\ .$$

Note that C_4 is a specific case of C_{4_i}, i.e. the case with $minPS = maxPS = 1$.

4.4.2 A symmetry breaker constraint

We assume that any two parties P_i and P_j which have same size, start time, and end time, are equivalent - i.e. if they are assigned P_i to table T_k and P_j to table T_h, then they can also be assigned P_i to table T_h and P_j to table T_k.

Let E be the set of all pairs of equivalent parties, i.e.:

$$E = \{(P_i, P_j) \mid i \neq j,\ (P_i.size, P_i.start, P_i.end) = (P_j.size, P_j.start, P_j.end)\}$$

We then define the symmetry breaker constraint:

$$C_{5_{ij}} = P_i < P_j, \quad \forall\, (P_i, P_j) \in E,\ i < j\ .$$

4.4.3 Example

Figure 4.8 shows our additional checks and constraints for the problem of Figure 4.3. C_3 ensures that the number of seats ($RC = 12$) is not less than the maximum number of diners seated at the same time ($maxNC = 9$, which happens at time $t = 1$). C_4 ensures that the number of usable tables ($M = 4$) is not less than the maximum number of parties overlapping in time ($maxNP = 3$, which happens at $t = 1$ and $t = 2$). C_{4_2} (as the original C_4) compares the maximum number of overlapping parties of size two or more to the total number of tables of size two or more; C_{4_3} compares the maximum number of overlapping parties of size three or more to the total number of tables of capacity three or more; and C_{4_4} compares the maximum number of overlapping parties of size four or more to the total number of tables of capacity four or more. Note that, if for example P_1, P_2, and P_3 were of size 4 the problem of Figure 4.3 would not be feasible as there is only one table for four. However, neither C_3, nor the original C_4 would spot the infeasibility, while C_{4_4} would find $(maxNP_4 = 3) \leq (M_4 = 1)$, which is false. For this example, C_3, C_4, and C_{4_i} are always true, but are shown here for illustration. Finally, C_5 breaks a symmetry in the problem, and ensures that an ordering is forced between pairs of equivalent parties ($E = \{(P_4, P_5)\}$). In the solution of Figure 4.3 (bottom) we have in fact $P_4 = 1 < P_5 = 3$.

Note that C_3 and C_4 represent a first feasibility check that the solver performs before starting the search. If any of them is violated then the problem is infeasible, then there is no need to continue with the search. $C_{5_{ij}}$ makes sure the search considers only one assignment over any set of symmetrical assignments. This constraint is expected to save a lot of search effort compared to the initial model, i.e. avoiding going through all equivalent assignments of values to variables, and therefore repeating many "symmetrical mistakes".

$$\begin{aligned}
C_3 &= maxNC \leq RC \ (maxNC = 9,\ RC = 12) \\
C_4 &= maxNP \leq M \ (maxNP = 3,\ M = 4) \\
C_{4_2} &= maxNP_2 \leq M_2 \ (maxNP_2 = 3,\ M_2 = 4) \\
C_{4_3} &= maxNP_3 \leq M_3 \ (maxNP_3 = 2,\ M_3 = 3) \\
C_{4_4} &= maxNP_4 \leq M_4 \ (maxNP_4 = 1,\ M_4 = 1) \\
\\
C_{5_{45}} &= P_4 < P_5
\end{aligned}$$

Figure 4.8: Additional checks and constraints for the example of Figure 4.3.

Example of symmetrical solutions - In Figure 4.9 we can see two solutions both valid but symmetric. Unless there is a preference, it is equivalent to consider one or the other allocation for parties P_4 and P_5. $C_{5_{45}}$ ensures only one symmetric solution is allowed, in this case the one at the bottom.

Table[size]	**0**	**1**	**2**	**3**
$T_1[2]$	P_1	P_1	P_5	P_5
$T_2[3]$		P_3	P_3	
$T_3[3]$			P_4	P_4
$T_4[4]$	P_2	P_2		

Table[size]	**0**	**1**	**2**	**3**
$T_1[2]$	P_1	P_1	P_4	P_4
$T_2[3]$		P_3	P_3	
$T_3[3]$			P_5	P_5
$T_4[4]$	P_2	P_2		

Figure 4.9: Two equivalent or symmetric seating plans: $P_4 > P_5$ (top) ; $P_4 < P_5$ (bottom).

4.5 CSP of the restaurant Eco with single tables

The specifics of the restaurant Eco were described in Section 2.3. In Eco, the range of possible start times on a dinner session is {4:00 p.m., 4:15 p.m., .., 10:00 p.m.}, i.e. 6:15 hours divided into 25 15-minute units, which we normalize to $R = \{\ 0, 1, .., 24\ \}$. Now we take the booking sheet represented in Figure 4.10 as example, and we define the corresponding CSP(X, D, C).

Party	Size	Start	End
P_1	5	1	5
P_2	2	1	5
P_3	3	2	6
P_4	2	2	8
P_5	5	3	8
P_6	4	4	11
P_7	4	4	10
P_8	4	4	9
P_9	4	4	8
P_{10}	3	4	9
P_{11}	2	4	12
P_{12}	3	5	12
P_{13}	5	6	10
P_{14}	4	6	13
P_{15}	4	6	12
P_{16}	2	6	11
P_{17}	2	6	10
P_{18}	5	7	12
P_{19}	3	7	12
P_{20}	2	7	13
P_{21}	6	8	16
P_{22}	3	8	16
P_{23}	3	8	14
P_{24}	3	8	13
P_{25}	2	8	16
P_{26}	2	8	15
P_{27}	2	8	13
P_{28}	5	9	13
P_{29}	3	9	17
P_{30}	3	9	15
P_{31}	2	9	14
P_{32}	4	10	17
P_{33}	3	11	19
P_{34}	6	12	17
P_{35}	4	12	18
P_{36}	3	12	16
P_{37}	2	12	20
P_{38}	3	13	21
P_{39}	2	13	22
P_{40}	2	13	21
P_{41}	2	13	19
P_{42}	3	14	20
P_{43}	2	14	22
P_{44}	2	15	25
P_{45}	5	16	25
P_{46}	4	17	26
P_{47}	4	17	25
P_{48}	3	17	26
P_{49}	3	17	26
P_{50}	2	17	27
P_{51}	2	17	26
P_{52}	2	17	24
P_{53}	4	19	26
P_{54}	4	19	25
P_{55}	3	19	29
P_{56}	3	19	28
P_{57}	2	20	28
P_{58}	6	21	29
P_{59}	6	21	27
P_{60}	4	21	29
P_{61}	2	21	26
P_{62}	4	22	31
P_{63}	2	22	28
P_{64}	5	23	30
P_{65}	4	23	29

Figure 4.10: Representation of a booking sheet of 65 parties - for convenience, parties have been listed in increasing order of start time.

Our set of decision variables is:

$$X = \{ \, P_1, P_2, .., P_{65} \, \} \, .$$

The set of possible values is:

$$Y = \{ \, T_1, T_2, .., T_{12}, T_{14}, .., T_{24}(orWT) \, \} \, .$$

Note that, in the restaurant Eco there is no table T_{13}, and we use T_{24} to represent the window table WT.

The set of initial domains is:

$$D = \{ \, D_1, D_2, .., D_{65} \, \} \, ,$$

$$D_i = \{ \, T_j \mid T_j.capacity >= P_i.size \, \} \, .$$

Depending on the party size, each decision variable has an initial domain as represented in Table 4.1.

Table 4.1: Domains in Eco.

Party size	**Initial domains**
1	$T_1, T_2, .., T_{12}, T_{14}, .., T_{24}$
2	$T_1, T_2, .., T_{12}, T_{14}, .., T_{24}$
3	$T_1, T_2, T_3, T_6, T_9, T_{10}, T_{11}, T_{14}, T_{15}, T_{16}, T_{17}, T_{24}$
4	$T_1, T_2, T_6, T_9, T_{10}, T_{11}, T_{14}, T_{15}, T_{16}, T_{17}, T_{24}$
5	$T_1, T_2, T_6, T_{11}, T_{14}, T_{15}, T_{16}, T_{24}$
6	$T_1, T_2, T_6, T_{14}, T_{15}, T_{16}, T_{24}$
7	T_{16}, T_{24}
8	T_{24}

There are 17 constraints C_{1_t}, defined for t in the subset of possible start times:

$$R^I \; = \{4, 5, 7, .., 17, 19, 20, 21, 23\} \, .$$

Then, $\{0, 1, 2, 3, 6\} \notin R^I$, because $X_0 \subset X_1 \subset X_2 \subset X_3 \subset X_4$, and $X_6 \subset X_7$. Similarly, $\{18, 22, 24\} \notin R^I$, as $X_{18} \subset X_{17}$, $X_{22} \subset X_{23}$, and $X_{24} \subset X_{23}$.

We now characterize constraint $C_{2_{khij}}$.

In Eco, any pair of parties of size 5 or 6 cannot simultaneously occupy either the pair of tables T_1 and T_{14}, or the pair T_2 and T_{15}, i.e.:

$$\begin{aligned} B = \{ \quad & (1,14,5,5),(14,1,5,5),(2,15,5,5),(15,2,5,5) \\ & (1,14,5,6),(14,1,5,6),(2,15,5,6),(15,2,5,6) \\ & (1,14,6,5),(14,1,6,5),(2,15,6,5),(15,2,6,5) \\ & (1,14,6,6),(14,1,6,6),(2,15,6,6),(15,2,6,6) \quad \} . \end{aligned}$$

The number of constraints $C_{2_{khij}}$ depends on the number of parties of size 5 or 6 which overlap in time.

For example, for the pair of parties (P_1, P_5) we have the 4 constraints:

$$\begin{aligned} C_{2\ 1\ 14\ 1\ 5} &= (P_1, P_5) \neq (1, 14) , \\ C_{2\ 2\ 15\ 1\ 5} &= (P_1, P_5) \neq (2, 15) , \\ C_{2\ 14\ 1\ 1\ 5} &= (P_1, P_5) \neq (14, 1) , \\ C_{2\ 15\ 2\ 1\ 5} &= (P_1, P_5) \neq (2, 15) . \end{aligned}$$

Further, in our example there are 18 pairs (i, j) of parties (P_i, P_j) of this type, i.e.:

$$\begin{aligned} (i,j) \in \ \{ \ \ & (1,5),\ (5,13),\ (5,18),\ (13,18),\ (13,21),\ (13,28) \\ & (18,21),\ (18,28),\ (21,28),\ (21,34),\ (28,34),\ (34,45) \\ & (45,58),\ (45,59),\ (45,64),\ (58,59),\ (58,64),\ (59,64) \ \} . \end{aligned}$$

Thus, as for each pair (i, j) there are 4 constraints $\{ C_{2\ 1\ 14\ i\ j}, C_{2\ 14\ 1\ i\ j}, C_{2\ 2\ 15\ i\ j}$, and $C_{2\ 15\ 2\ i\ j} \}$, the total number of $C_{2\ k\ h\ i\ j}$ is 72.

Finally, we redefine the additional checks and constraints C_3, C_{4_i}, and C_5.

The maximum capacity of the restaurant Eco using single tables is $RC = 83$. Further, the maximum number of covers at any time in the booking problem is:

$$maxNC = \sum_{i=1..N,\ P_i.table\ \in\ X_{23}} P_i.size \ = \ 76\ .$$

Therefore:

$$C_3 \ = maxNC \ \leq RC\ .$$

The number of tables in the restaurant Eco is 23, all 23 can accommodate at least 2 or more, 12 can accommodate at least 3, 11 at least 4, 8 at least 5, 7 at least 6, 2 at least 7, and 1 can seat 8 (i.e. $maxTC = 8$). Then, as the sizes of the parties in our booking sheet ranges from 2 to 6, we have:

$$\begin{aligned}
C_{4_2} &= maxNP_2 \leq M_2\ (M_2 = 23)\ , \\
C_{4_3} &= maxNP_3 \leq M_3\ (M_3 = 12)\ , \\
C_{4_4} &= maxNP_4 \leq M_4\ (M_4 = 11)\ , \\
C_{4_5} &= maxNP_5 \leq M_5\ (M_5 = 8)\ , \\
C_{4_6} &= maxNP_6 \leq M_6\ (M_6 = 7)\ .
\end{aligned}$$

To conclude, $C_{5_{ij}}$ is defined on the set of equivalent parties, which in our example is composed by only a single pair $E = \{(P_{48}, P_{49})\}$, i.e.:

$$C_{5\ _{48\ 49}} \ = \ P_{48} < P_{49}\ .$$

4.6 Multiple heuristics and time-slicing

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.

4.6.1 Initial tests on single heuristics

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. Table 4.2 shows a representative example of our initial results, reporting the run-time of 3 different heuristics over 4 problem instances. For this example, the time limit we imposed on each solution process was 40 seconds. As we can see, none of the heuristics can solve each of the instances within the time limit, but for each instance there is at least one heuristic which can solve it.

Table 4.2: r-time for different heuristics over the same set of instances.

heuristic	**instance 1**	**instance 2**	**instance 3**	**instance 4**
$h1$	$0.07sec$	$> 40sec$	$0.05sec$	$> 40sec$
$h2$	$> 40sec$	$> 40sec$	$> 40sec$	$0.21sec$
$h3$	$0.04sec$	$0.16sec$	$0.65sec$	$> 40sec$

4.6.2 Using multiple heuristics

We devised a restart approach with multiple different ordering heuristics, and an increasing time limit for each set of restarts. The aim is to exploit the variance among the orderings to get a more robust procedure, which may be slower on some problems, but avoid the significant deterioration on others. Now we describe the details of this multi-heuristic algorithm (MH), and later in the chapter we will demonstrate the benefit, in terms of efficiency and robustness, of the approach. The pseudocode for the algorithm is shown in Figure 4.11.

Solve(*heuristic(i),limit*) 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(*i,limit*) is the time limit function, and we have considered the two versions below: *(a)* increases *limit* by δ each time *i* is incremented (i.e. linearly); *(b)* increases *limit* by one order of magnitude every *n* loops (e.g. *n* = 10).

$$\begin{array}{rrcl}
\text{a)} & \textbf{linear} & : & increase(i, limit) = limit + \delta \\
\text{b)} & \textbf{magnitude} & : & if\ (i == n)\ increase(i, limit) = limit * 10 \\
 & & & else\ \quad increase(i, limit) = limit
\end{array}$$

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 [64] explores each branch to a certain depth, and then increases the depth limit, and is similar to randomized restarts [51], except we use a set of systematic ordering heuristics.

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

Figure 4.11: Multiple-heuristic algorithm.

4.6.3 Properties

Proposition 4.6.3.1. *MH is complete.*

Proof. The CSP backtracking search space is finite, each ordering heuristic is systematic, and *limit* increases indefinitely, so eventually one of the heuristics will be given enough time to complete the search. □

Proposition 4.6.3.2. *MH has a performance guarantee. For either of the time limit functions,* magnitude *or* linear*, if any one of the heuristics is deterministic then MH has a guaranteed upper bound on the ratio of the time it takes to solve compared to the time that heuristic takes to solve.*

Proof. Let $\{h_1, h_2, .., h_H\}$ be the ordered set of heuristics used by the multiple heuristics algorithm, i.e. h_H is the last heuristic tried in each round of restarts. As at least one of the H heuristics is deterministic, we can assume h_H is deterministic. We call t_H and t_{MH} the time to solve for h_H and MH respectively. Then, we have to study the upper bound of the performance ratio $PR = t_{MH}$ / t_H.

A typical set for our experiments was $H = 33$ and $limit_0 = 0.01$sec, where $limit_0$ is the initial value taken by $limit$.

<u>*Magnitude limit function*</u>

Lets assume we are using MH with a limit function of type *magnitude*.

If h_H can solve within $limit_0$, i.e.:

$$0 < t_H \leq limit_0 \ ,$$

then MH can solve within the first round of restarts:

$$0 < t_{MH} \leq limit_0 \times H \ ,$$

and therefore, with $limit_0 = 0.01$sec and $H = 33$, we obtain:

$$0 < t_{MH} \leq 0.33 \ sec \ .$$

Otherwise, let $k \geq 1$ be an integer such that:

$$limit_0 \times 10^{k-1} < t_H \leq limit_0 \times 10^k \ .$$

MH contains h_H, therefore MH would succeed to solve during the $(k+1)^{th}$ round of restarts at the latest, i.e. as other heuristics in MH could solve in earlier rounds.

For example, if:

$$limit_0 < t_H \leq limit_0 \times 10 \ ,$$

then MH can solve within the second round, when $limit = limit_0 \times 10$.

H is the cardinality of the set of heuristics, hence:

$$t_{MH} \leq limit_0 \times H + limit_0 \times 10 \times H + ... + limit_0 \times 10^k \times H =$$

$$= limit_0 \times H \times (10^0 + 10^1 + ... + 10^k) =$$
$$= limit_0 \times H \times \sum_{n=0..k} 10^n \ .$$

The last factor is the geometric (or exponential) series, for which we known that:

$$\sum_{n=0..k} 10^n = \frac{10^{k+1} - 1}{10 - 1} \leq \frac{10}{9} \times 10^k \ , \ \ i.e.$$

$$\sum_{n=0..k} 10^n = O(10^k) \ .$$

Then we can write:

$$t_{MH} < limit_0 \times H \times \frac{10}{9} \times 10^k \ .$$

To define the upper bound of the performance ratio PR, we assume that:

$$t_h = limit_0 \times 10^{k-1} + \ \epsilon \ , \ \ with \ \ \epsilon \ \ small \ .$$

Thus, the upper bound can be defined as:

$$PR = \frac{t_{MH}}{t_H} < \frac{limit_0 \times H \times \frac{10}{9} \times 10^k}{limit_0 \times 10^{k-1}} \ , \ \ i.e.$$

$$PR < H \times \frac{10^2}{9} \ \ (e.g. \ PR < 367 \ \ for \ H = 33) \ .$$

Linear limit function

Lets now assume we are using MH with a limit function of type *linear*. A typical set for our experiments based on the *linear* function is $\delta = limit_0 = 0.01$sec.

If h_H can solve within $limit_0$, i.e.:

$$0 < t_H \leq limit_0 \ ,$$

then MH can solve within the first round of restarts:

$$0 < t_{MH} \leq \sum_{i=0..H-1} (limit_0 + (\delta \times i)) \ ,$$

$$0 < t_{MH} \leq (H \times limit_0) + \delta \times \sum_{i=0..H-1} i \ .$$

The arithmetic series gives:

$$\sum_{i=1..N} i = \frac{(N+1) \times (N)}{2} \ .$$

Then we obtain:

$$0 < t_{MH} \leq (H \times limit_0) + \delta \times \frac{H \times (H-1)}{2} \ ,$$

$$0 < t_{MH} \leq H \times \left(limit_0 + \frac{\delta \times (H-1)}{2} \right) \ ,$$

and therefore, with $limit_0 = \delta = 0.01$sec and $H = 33$, we obtain:

$$0 < t_{MH} \leq H \times limit_0 \left(1 + \frac{H-1}{2} \right) \ ,$$

$$0 < t_{MH} \leq 33 \times 0.01 \times 17 \ ,$$

$$0 < t_{MH} \leq 5.61 \ sec \ .$$

In general, let $k \geq 0$ be an integer such that:

$$limit_0 + \delta \times (kH - 1) < t_H \leq limit_0 + \delta \times ((k+1)H - 1) \ .$$

MH contains h_H, therefore MH would succeed to solve during the $(k+1)^{th}$ round of restarts at the latest, i.e. as other heuristics in MH could solve in earlier rounds.

For example, if

$$limit_0 + \delta \times (H-1) < t_H \leq limit_0 + \delta \times (2H-1) \ ,$$

then MH can solve within the second round of restarts.

H is the cardinality of the set of heuristics, hence:

$$\begin{aligned} t_{MH} \leq \quad & limit_0 + \delta + 2\delta + ... + (H-1)\delta + \\ & + H\delta + (H+1)\delta + (H+2)\delta + ... + (2H-1)\delta + \\ & ... \\ & + (k-1)H\delta + (k-1)(H+1)\delta + (k-1)(H+2)\delta + ... + (kH-1)\delta \ , \end{aligned}$$

i.e.:

$$\begin{aligned} t_{MH} \leq limit_0 + \delta \times (1 + 2 + ... + (kH-1)) = \\ = limit_0 + \delta \times \sum_{n=1..kH-1} n \ . \end{aligned}$$

The arithmetic series gives:

$$\sum_{n=1..N} n = \frac{(N+1) \times N}{2} \ .$$

Then:

$$t_{MH} \leq limit_0 + \delta \times \frac{kH \times (kH-1)}{2} \ .$$

To define the upper bound of the performance ratio, we assume that:

$$t_h = limit_0 + \delta \times (kH-1) + \ \epsilon \ , \ \ with \ \ \epsilon \ \ small \ .$$

Thus, the upper bound can be defined as:

$$PR = \frac{t_{MH}}{t_H} < \frac{limit_0 + \delta \times \frac{kH \times (kH-1)}{2}}{limit_0 + \delta \times (kH-1)} \ , \ \ i.e.$$

$$PR = \frac{t_{MH}}{t_H} < \frac{kH}{2} \; with\, limit_0 \; small \; , \; (e.g. \; PR < \frac{k \times 33}{2} \; for H = 33) \; .$$

□

4.6.4 MH configuration

In total, we selected 11 different variable ordering heuristics, and 3 different value orderings, giving a total of $H = 33$ different heuristic combinations.

Variable orderings - We utilized the list of variable (or party) orderings represented in Table 4.3. $H1$ and $H2$ are two standard versions of min-size domain. For example, in Figure 4.12 P_2 has the smallest domain (or set) of suitable tables $D_2 = \{T_4\}$, therefore $H1$ and $H2$ would assign this party first. In case of ties, $H1$ selects randomly while $H2$ picks the first party in lexicographical order. In the example, P_1 and P_4 have the same domain size, thus $H2$ would assign P_1 first. $H3$ to $H10$ are static orderings created from sorting the set of parties by start time and size. For example, $H10$ sorts by decreasing party size, breaking ties by decreasing start time - i.e. it would chronologically assign P_2, P_3, P_4, and finally P_1. Finally, $H11$ involves a measure of resource contention [4] among parties, i.e. it sorts by counting, for each party, the number of other parties which overlaps in time. Thus, in Figure 4.12, party P_3 would count 3 (overlapping with P_1, P_2, and P_4), P_1 and P_2 would count 2, and P_4 would count 1, so P_3 would be tried first.

Value orderings - We utilized the list of value (or table) orderings represented in Table 4.4, including three static orders. The first two choose among tables with the smallest or highest (suitable) capacity first. The third consists of a random table order - i.e. we use the numerical order of tables (T_1, T_2, .., etc.). Note that in the example of Figure 4.12 the capacity of the four tables increases with the table number, therefore W1 and W3 represent the same order. However, this is not the case for the problem based on the Eco restaurant, where the list of tables T_1, T_2, .., T_{24} is not ordered by increasing capacity.

MH versions - We combined both lists of variable and value heuristics together, implementing four MH versions: MH(11×3), MH(11×1), MH(1×3), and MH(1×1). All have $H1$ and $W1$ as first variable and value heuristics. MH(11×3) is the full version, using the 11 variable orderings and the 3 value orderings com-

Table 4.3: List of variable (or party) ordering heuristics.

Heuristic id	Ordering	Tie breaking
H1	min-size domain	random
H2	min-size domain	lexicographic
H3	increasing start time	increasing size
H4	increasing start time	decreasing size
H5	decreasing start time	increasing size
H6	decreasing start time	decreasing size
H7	increasing size	increasing start time
H8	increasing size	decreasing start time
H9	decreasing size	increasing start time
H10	decreasing size	decreasing start time
H11	most overlapping in time	lexicographical

Table 4.4: List of value (or table) ordering heuristics.

Heuristic id	Ordering	Tie breaking
W1	increasing capacity	lexicographic
W2	decreasing capacity	lexicographic
W3	arbitrary fixed order	

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

Table[size]	0	1	2	3
$T_1[2]$	P_1	P_1	P_4	P_4
$T_2[3]$		P_3	P_3	
$T_3[3]$				
$T_4[4]$	P_2	P_2		

Figure 4.12: Problem instance (top); and a possible seating plan (bottom).

bined in the 33 possibilities. MH(11×1) uses the 11 variable orderings combined with the first value ordering. MH(1×3) uses the first variable ordering combined with the 3 value orderings. Finally, MH(1×1) is the single heuristic version, based on the combination of $H1$ and $W1$.

4.6.5 Evaluation metrics: efficiency and robustness

Our current problem is a feasibility problem, i.e. we want to answer questions like (i) given the current booking sheet, can we fit a new booking of 4 people at 6 p.m., even if this would require reallocating all the other parties? or (ii) when a dinner lasts longer than expected, can we reallocate all future parties without delaying anybody? Note that, at the moment we do not care about how good a seating plan is, i.e. in (i) and (ii) we consider any reallocation, for example even those that would assign many six-seater tables to parties of two people.

Efficiency

For our feasibility problem, efficiency is an important metric for evaluating how good the CSP-solving procedure is. For example, given a problem instance of type (i) or (ii), a solution available in half an hour is not practically worthwhile (or usable). For simplicity, we can classify a solving process as efficient if the time before returning is between 10 to 20 seconds - of course, the quicker the response becomes available the better.

Robustness

The second important metric is robustness, which can be regarded as efficiency averaged (or distributed) over different problem instances. A solver which solves most problem instances in reasonable time has to be preferred to one which can solve some instances very quickly (perhaps much more quickly than the first solver) but can be too slow for many others. The first solver represents a more robust solving procedure.

4.6.6 Experiments

We want to test our final model and see whether it can represent in practice an efficient and robust solution. Specifically, we want to test the benefit of the additional constraints and the performance of the time-sliced multi-heuristic approach. The questions we want to answer are:

(i) how do pre-solving checks, additional constraints, and MH speed up the search, i.e. how do extra checks and constraints save the search from unnecessarily exploring wrong portions of the search tree, and how does MH improve over using any of its heuristic components singularly?

(ii) is MH more robust than the standard default ordering heuristic, i.e. does it report a result within acceptable time limits in more cases across a range of problems?

(iii) does MH avoid a significant increase in run time, i.e. is the overhead of restarting the search, and repeating some search paths, significant?

(iv) how does MH compare to the randomized restart method, i.e. is its performance due to the restart mechanism, or to the multiple heuristics?

To answer these questions, we have tested the approach on two problem classes: on our restaurant scheduling problem, with single tables and with fixed start and end times, and on quasi-groups with holes (QWH). All implementations are coded in C++ using Ilog Solver 6.0, and run on a Pentium 2.6 GHz processor under Linux.

For a first answer to question (i) we test over the original scheduling problem of 23 tables. We consider a model based on MH search, and compare between the versions with and without additional constraints. Then we consider a model using the additional constraints, and compare between versions using single heuristics and a version using MH.

For the last three questions, we extend our tests over a larger scheduling problem concerning a restaurant of 100 tables, and then over the class of QWH problems. We compare a model based on MH against models using the recommended heuristics. For instance, for (ii) and (iii) we compare MH against the min-size domain (*msd*) variable ordering heuristic (with lexicographic tie breaking), and for (iv) we compare against the same variable ordering heuristic but with random tie breaks, and random value ordering[†].

4.6.7 Test dimensioned upon the restaurant Eco

We generate random booking sheets. The size (or number of parties) of the booking sheets ranges from 10 to 100 in single steps, for a total of 91 different sizes. Start times are distributed from 4 p.m. to 10 p.m. over 25 time units of 15 minutes each - the interval has been normalized to take integer values in the range [0..24]. We consider different start time distributions (over this interval). We also consider constant (6 time units), uniform (4 to 8 time units), or realistic (2 to 10 time units) distributions of dinner durations. Finally, the party size is generated from a uniform distribution [1..8], where 8 is the maximum party size a single table can accommodate. We perform tests over 1000 instances for each possible size of booking sheet, for a total of 91000 problems for each distribution of start time and duration.

Limit function and time limit - The search algorithm based on MH performs a first and quick run over the set of heuristics, switching heuristic every 0.02 second, and then a second run, switching every 0.5 seconds. We impose the time-limit for each run to 17 seconds, after that the problem is regarded as unsolved. The time-limit was tuned to 17 seconds to allow the multi-heuristic algorithm to complete the two runs over the 33 combinations of variable and value orderings. It also represents a reasonable time window for a practical use in a restaurant.

Evaluating the additional checks and constraints

Figure 4.13 compares the mean running time of a model using the essential constraints, CSP(X, D, C1, C2), to the same model extended with the additional (or implied) checks and constraints, CSP(X, D, C1, C2, C3, C4, C5). The two models

[†]We use the min-domain heuristic for randomized-restarts as opposed to other heuristics (e.g. Brelaz) because min-domain is the ordering heuristic recommended by Ilog, which provides it as a default function. The Ilog function that implements min-domain was also used by Gomes et. al. [49] [50] in their original work on randomizing restarts to solve a class of scheduling problems.

combine the (basic or extended) constraint representations with a search based on multiple heuristics. The left hand axis has log scale resolution.

Solvability - The dashed curve from top left to bottom right represent the solvability, i.e. the percentage of instances which have been found feasible (note that each point in the graph is an average over 1000 instances). As the number of parties per booking sheet increases, the solvability decreases. We can see that all instances with more than 82 parties allow no solution, while about 50% of instances with 60 parties are feasible.

Essential constraints - We can see that the run-time for the version without implied constraints increases significantly as the problem grows in size. In particular, for booking sheets of more than 75 parties, i.e. where the solvability is approaching zero, the time gets to 17 seconds. In other words, after that point the model cannot solve any instance within the time limit. This curve profile means that without the additional checks and constraints, when a problem is infeasible the model needs to entirely explore a large search tree in order to prove no solution exists. For the size of our problem this exploration cannot be done in reasonable time, i.e. within the 17 seconds.

Additional checks and constraints - The model extended with the implied checks and constraints performs much better, with the mean run-time never exceeding a small fraction of a second. The benefit of the model extension can be explained in terms of help to immediately spot infeasibility, and in terms of search effort reduction. For instance, all those instances which exceeds the restaurant capacity (i.e. 83 covers, 23 parties of 2 or more, 12 parties of 3 or more, .., 1 party of 8 or more) violate some of the pre-solving checks, therefore for them no search is required to prove no solution exists. Further, the symmetry constraint reduces the size of the search tree. Therefore, even in case of infeasible instances which do not violate the capacity pre-checks, the search effort necessary to go through all possible allocations is much more contained.

Multiple versus single heuristics

Figure 4.14 compares the performance of models adopting a search based on a single heuristic to a model adopting a search based on multiple heuristics. All

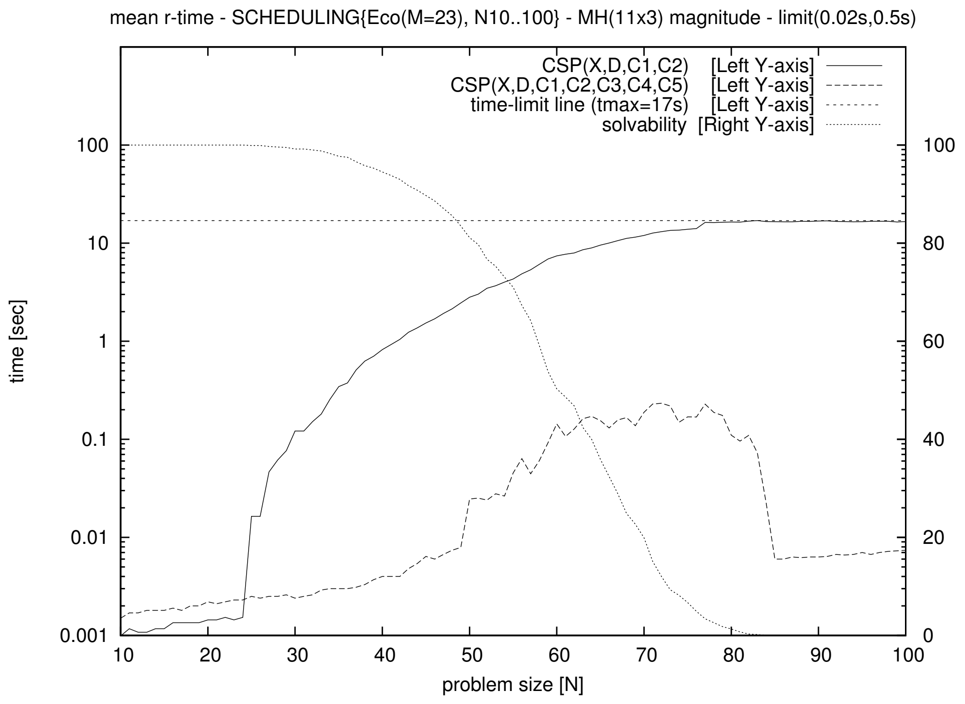


Figure 4.13: Comparison of CSP(X, D, C1, C2) (model with basic constraints) against CSP(X, D, C1, C2, C3, C4, C5) (model extended with additional checks and constraints). Each point is an average over 1000 instances. Note log-scale on left hand axis. The solvability curve refers to the right hand axis. All instances with more than 82 parties were found infeasible.

models use the extended constraint representation including the additional checks and constraints. Again, each point in the graph is an average over 1000 instances.

Single heuristics - We can see that the run-time is different for different heuristics. The worst performance comes from using a random variable ordering and a value ordering which is the lexicographical order of the tables in the Eco restaurant. The peak of difficulty for such heuristic is located around 55 to 60 parties, where the time is about 9 seconds on average. The second worst heuristic is based on selecting parties by decreasing start time (breaking ties by decreasing size), and on assigning tables by decreasing capacity. A better performance comes from the third heuristic, which chooses the parties by increasing time (breaking ties by decreasing size) and assigns tables by increasing capacity. We expect that this improvement is mostly due to the value ordering. In fact, the previous heuristic considers a party of two allocated into a table for 6 before trying it into a smaller table, and this strategy is more likely to waste time exploring many unsuccessful paths of the search tree before eventually reaching a valid solution. Finally, the best performance using a single heuristic comes from the min-size domain heuristic, which is the ordering recommended by the constraint programming community. Such heuristic selects the next party choosing the one with the minimum domain (or number of suitable tables) left. The ordering given by this heuristic is dynamic, i.e. the domains are recomputed after each value (or table) assignment has been propagated through all the remaining variables. The peak of difficulty is significantly reduced compared to the first (and worst) heuristic - it went from around 9 to around 0.9 seconds (i.e. one order of magnitude less). The peak has also shifted closer to the infeasible zone - moving from around 55 to around 70 parties.

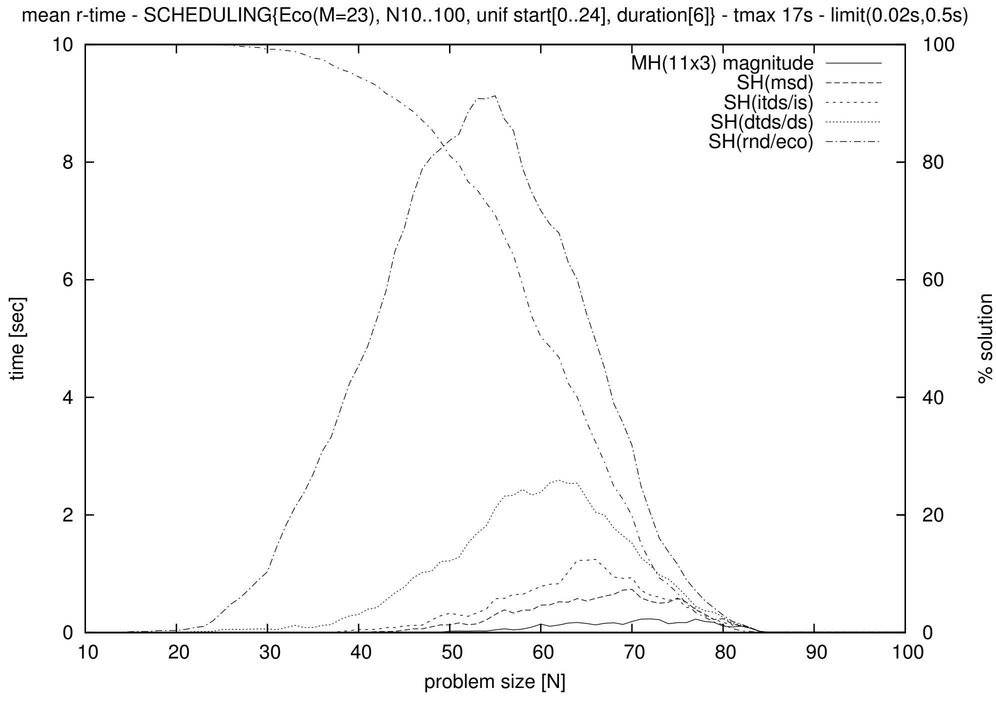


Figure 4.14: Benefit of using multiple rather than single heuristics.

Multiple heuristics - The curve concerning our multiple heuristics algorithm is the one at the bottom. Using this search method we obtain another significant improvement, with a peak of difficulty now reduced to less than 0.3 seconds, which means we improved another 66% compared to the performance of the min-size domain heuristic. Note that the difficult instances are located around the end of the transition from complete solvability to complete insolvability, i.e. where most of them are infeasible.

Constant versus uniform dinner duration

We consider a first test assuming all dinners have a constant duration of six 15-minute units (i.e. 1:30 hours), and a second test where duration have been generated randomly from a uniform distribution in the range [4..8], i.e. from 1:00 hour to 2:00 hours. Figure 4.15 compares the results obtained. Both tests have been performed using the model extended with the additional checks and constraints, and based on MH. As we can see, the two curves are similar, perhaps the one with constant durations is less noisy, however both peaks are only slightly over two tenths of a second.

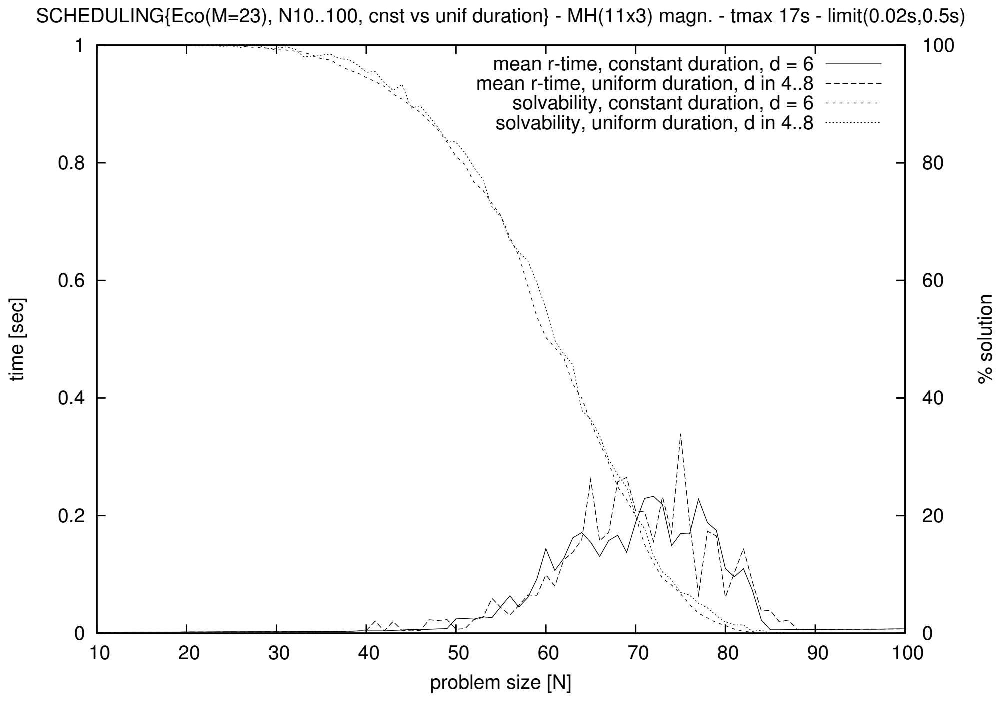


Figure 4.15: Results on constant or uniform dinner durations.

Realistic start times and dinner durations

We now consider a test where start times and dinner durations are distributed more realistically (Figure 4.16 and 4.17). For example, in a standard evening, dinners may last longer in the period from 7 o'clock to 9:30 p.m. - earlier people tend to seat for a quicker meal, while later people arrive knowing the kitchen is going to close soon. Even the pattern for the start times can be non-uniform, e.g. there can be a peak of bookings at around 6 o'clock and another at around 8:30 p.m., and in between there can be a hole because many people may go to the theater.

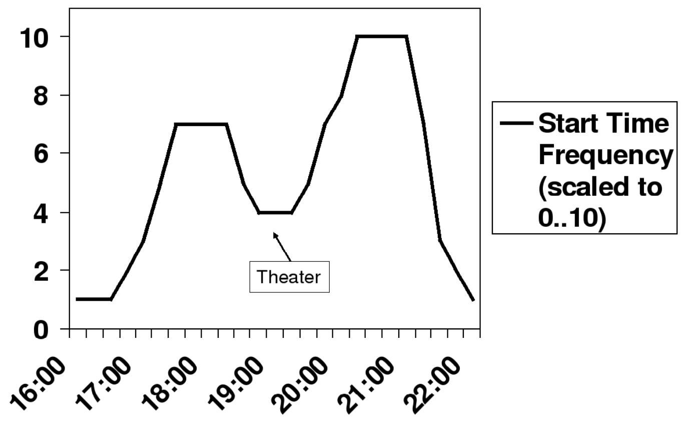


Figure 4.16: Realistic distribution of start times.

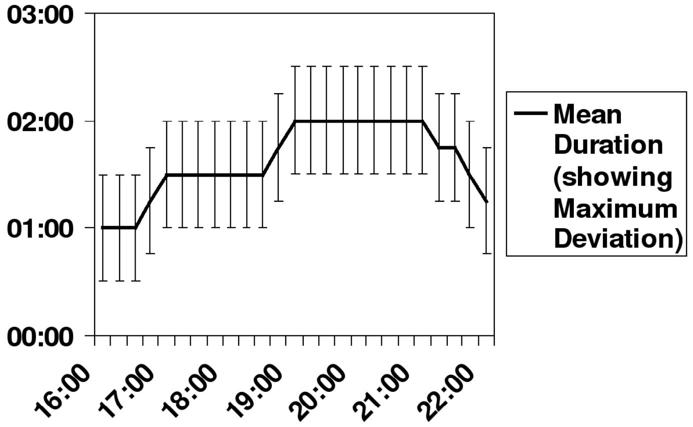


Figure 4.17: Realistic distribution of dinner durations. The curve shows the mean duration over time. For each point in the curve, durations are uniform within the range (or maximum deviation) represented.

Figure 4.16 shows the start time frequency, scaled between 0 and 10, over time. In Figure 4.17, the curve represents the mean duration over time. For each point in the curve, the maximum deviation from the mean is half an hour. Durations are uniform within the range (or maximum deviation) represented - e.g. dinners starting at 4 p.m. have durations uniformly distributed between half an hour and 1:30 hours, while at around 8 p.m. the duration ranges uniformly from 1:30 hours to 2:30 hours.

Figure 4.18 shows the performance of our model over problems with start time and duration distributed as in Figure 4.16 and 4.17. The party size is generated uniformly in the range 1 to 8. We can see that the model is again very efficient over the entire horizontal axis, with a peak of 0.2 seconds located at around 50 parties per problem. Comparing to the previous distributions the solvability transition (curve from top left to bottom right) happens earlier, which means that with a less even distribution the restaurant can allocate fewer parties on average. Note that the difficult instances are still located around the end of the transition, i.e. where most of them are infeasible.

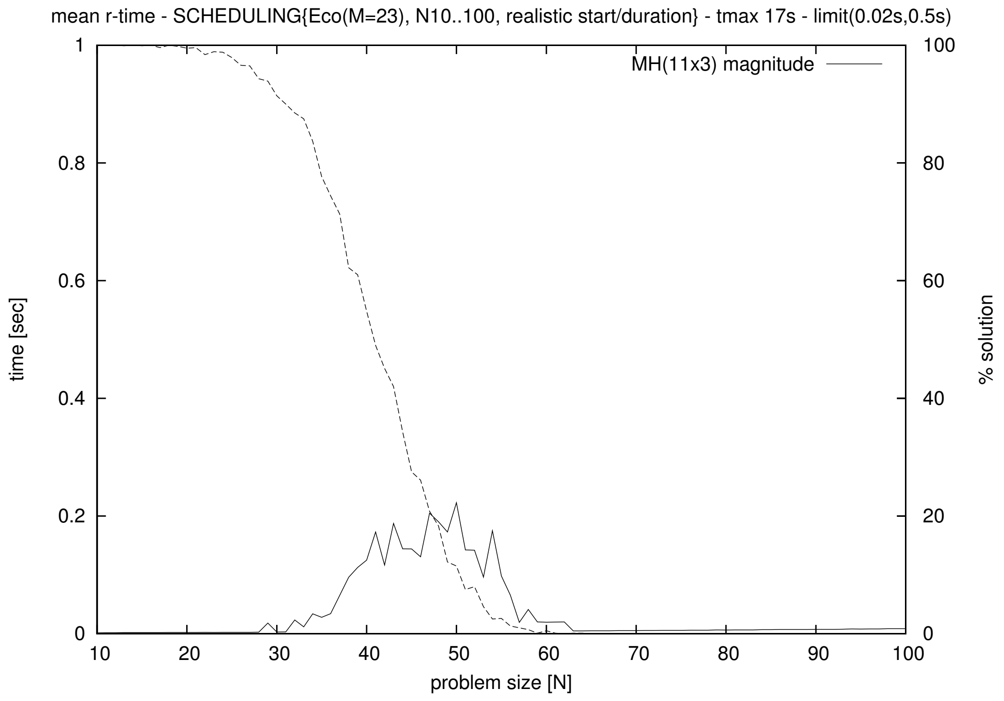


Figure 4.18: Results on realistic start times and dinner durations.

First conclusions

Using the final model configuration, i.e. extending the basic model with the additional checks and constraints, and with the multiple heuristics search, we have significantly improved the performance and we are now able to solve the static problem efficiently. Instances representing a full booking sheet of 60 parties (or 200 covers) can be solved in a small fraction of a second on average. 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.

4.6.8 Tests scaled on a restaurant of 100 tables

In this section, we want to test whether our model can provide efficient solutions also to restaurants of larger size. We consider one set of test problems, $\langle$ 100, 10, N $\rangle$, with 100 restaurant tables uniformly distributed over 10 possible capacities [1..10]. We varied the number of parties, N, from 130 to 200 (in single steps), and for each one we generated 500 random problems, choosing start times in [0..40], durations in [17..25] and size in [1..10], all uniformly at random. For each instance, we impose a maximum time of 41 seconds, which allows time slices of 0.01, 0.1, and 1 second for 33 possible heuristics, including the overhead on initializing the problem.

Comparing to the standard recommended heuristic (msd)

The table at the top of Figure 4.19 shows the number of times MH(1×1) (or min size domain) and MH(11×3) hit the time limit, while the graph at the bottom reports the mean run time of the 4 versions MH(11×3), MH(11×1), MH(1×3), and MH(1×1). MH in its full version MH(11×3) consistently outperforms and improves msd. It is more robust - it hits the time limit on fewer occasions. It also has a lower mean run time across the range. We can also observe that, as we start introducing more than one value heuristic, i.e. MH(1×3), we obtain a first clear improvement. There is benefit also by including more than one variable heuristic, i.e. MH(11×1). Note that passing from MH(1×1) to MH(11×1) the majority of the variable heuristics we add to the dynamic msd are all static. Things get better again when combining all the variable heuristics with all the value heuristics, i.e. MH(11$\times$3). Recall that the line on the graph from top left to bottom right shows solubility, and relates to the right hand axis - e.g. almost 50% of size 150 problems have a solution. The hardness peak is where most problems have no solution.

failure frequency [%]		
Size [N]	min size domain	MH magnitude
130	10	4
140	22	12
150	62	16
160	58	32
170	82	40
180	28	22
190	2	0
200	0	0

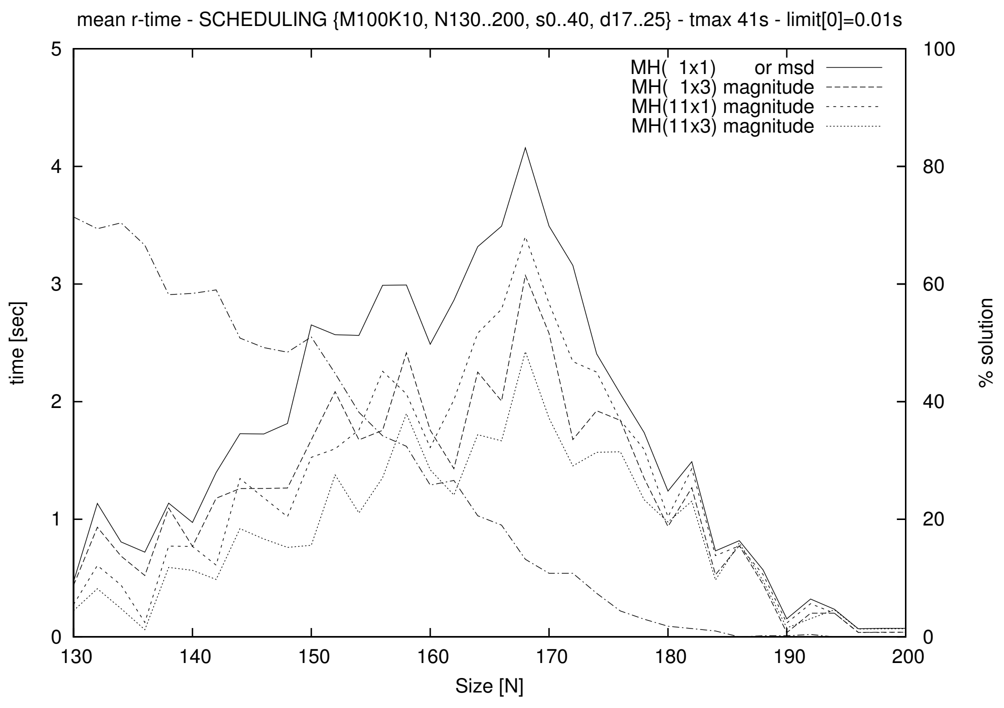


Figure 4.19: Comparing MH against msd: frequency of failure to solve within *tmax* (top); mean r-time (bottom).

On other tests we also saw that excluding msd from the full version MH(11×3) has a small effect on performance. Figure 4.20 reports the comparison. Specifically, we compared against MH(10×3) where we utilized the list of variable orderings represented in Table 4.5, combined with the original list of value orderings represented in Table 4.4. Note that Table 4.5 is obtained from the original list in Table 4.3, by excluding $H1$ and $H2$, i.e. the two versions of min-size domain, and by including a fixed random variable ordering ($H12$).

Table 4.5: New set of variable ordering heuristics without msd.

Heuristic id	**Ordering**	**Tie breaking**
H3	increasing start time	increasing size
H4	increasing start time	decreasing size
H5	decreasing start time	increasing size
H6	decreasing start time	decreasing size
H7	increasing size	increasing start time
H8	increasing size	decreasing start time
H9	decreasing size	increasing start time
H10	decreasing size	decreasing start time
H11	most overlapping in time	lexicographical
H12	fixed random ordering	-

Comparing to randomized-restarts on min domain

We compare MH(11×3) against the randomized restart approach (RR) introduced in Chapter 3. We implement RR using msd as variable ordering heuristic (with randomized tie-breaking), and using a random value ordering heuristic. RR is generally used with time limits that increase with each restart, so we implement MH with the same time policy as standard RR, and RR with an order of magnitude time increased every $n = 33$ restarts, for comparison. The two deepening mechanisms, *linear* and *magnitude*, were discussed in Section 4.6.

In Figure 4.21, MH clearly improves on RR at the peak of difficulty, which is located in the region where approximately 90% of instances have no solution. The gap is present for both time-slicing versions, i.e. increasing linearly every single restart (MH-linear vs. RR-linear), and increasing by an order of magnitude

failure frequency [%]		
Size [N]	MH magnitude no msd	MH magnitude with msd
130	6	4
140	15	12
150	23	16
160	32	32
170	49	40
180	22	22
190	0	0
200	0	0

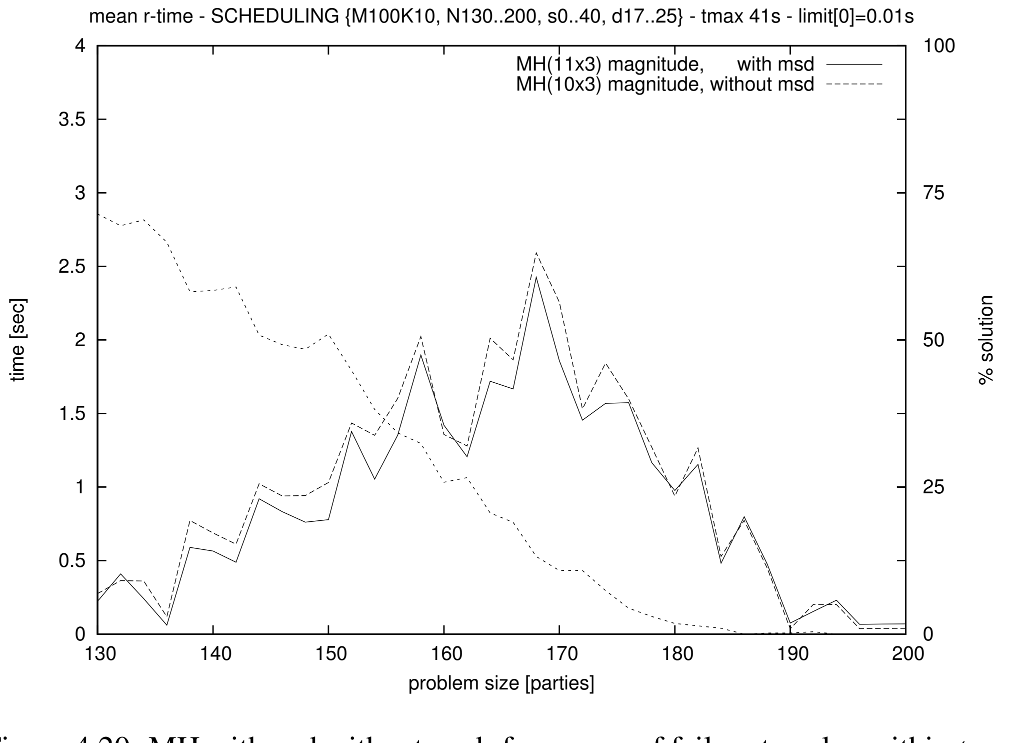


Figure 4.20: MH with and without msd: frequency of failure to solve within *tmax* (top) mean r-time (bottom).

every loop (MH-magnitude vs. RR-magnitude). There is actually only a slight difference between the two time-slicing versions, with the magnitude mechanism better on average.

failure frequency [%]		
Size [N]	RR magnitude	MH magnitude
130	2	4
140	14	12
150	18	16
160	42	32
170	62	40
180	32	22
190	0	0
200	0	0

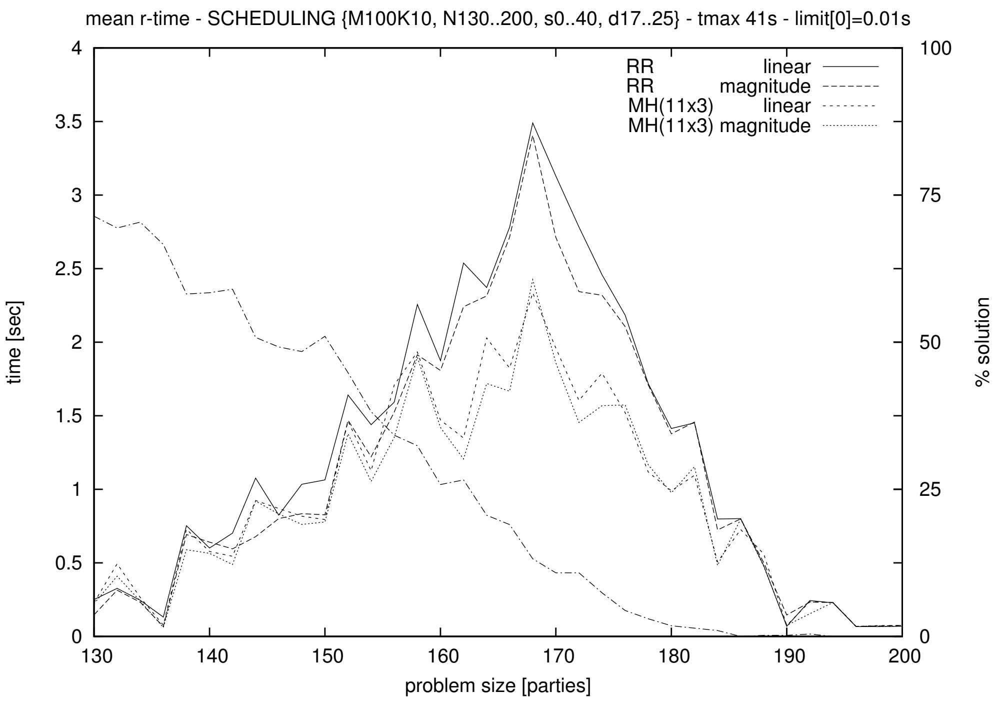


Figure 4.21: Comparing MH against RR: frequency of failure to solve within *tmax* (top); mean r-time (bottom).

4.6.9 Tests on quasi-group with holes (QWH)

A quasi-group of order N is a Latin Square of N by N cells. The solution of a Latin Square requires an allocation to each cell of a number from 1 to N, so that all the elements appearing on each row are different and all the elements appearing on each column are also different. A quasi-group with holes (QWH) [1] [51] is a solved Latin Square from which some allocations are deleted. The problem is to find an allocation which completes the Latin Square.

CSP representation - In our CSP model, we represent the empty cells as variables, and the numbers as the values to be assigned. We use the global constraint *all-different* to ensure each row and column has allocations that are all different. In Figure 4.22 we represent: a problem instance of QWH($N = 4$) with $H = 13$ holes (top); the remaining domains (center); and a possible solution (bottom).

1		2	
	2		

1	3,4	2	3,4
3,4	2	1,3,4	1,3,4
2,3,4	1,3,4	1,3,4	1,2,3,4
2,3,4	1,3,4	1,3,4	1,2,3,4

1	3	2	4
3	2	4	1
2	4	1	3
4	1	3	2

Figure 4.22: QWH: an instance (top), remaining domains (middle), and a solution (bottom).

Variable orderings - We utilized the list of variable orderings represented in Table 4.6. $H1$ and $H2$ are two standard versions of min-size domain. $H3$ to $H10$ are static orderings created from sorting the cells by column and row.

Value orderings - We utilized the list of value orderings represented in Table 4.7, which includes three orders, two static and one dynamic. $W1$ ($W2$) simply chooses smallest (biggest) numbers first. $W3$ involves a measure of conflict among numbers: if variable X is chosen, $W3$ looks the number frequency in the domains of all the unassigned variables in the same row and column as X. Knowing that all numbers must appear once in the column and once in the row $W3$ choose the number that appears least in domains of the other unassigned variables in the row and column. Thus, in the example of Figure 4.22 above, assuming the bottom-right variable is chosen first, value 1 would count 4, 2 would count 2, 3 and 4 would count 6, so $W3$ would choose value 2 first.

Table 4.6: List of variable ordering heuristics.

Heuristic id	**Ordering**	**Tie breaking**
H1	min size domain	random
H2	min size domain	lexicographic
H3	increasing column	increasing row
H4	increasing column	decreasing row
H5	decreasing column	increasing row
H6	decreasing column	decreasing row
H7	increasing row	increasing column
H8	increasing row	decreasing column
H9	decreasing row	increasing column
H10	decreasing row	decreasing column

Table 4.7: List of value ordering heuristics.

Heuristic id	**Ordering**	**Tie breaking**
W1	min number first	lexicographic
W2	max number first	lexicographic
W3	least (x, y)-conflicted number	lexicographic

MH version - We combined the lists of variable and value heuristics, implementing MH(10×3), where the first combination tried by MH is (H1×W1).

Test setting

Experiments regarded balanced QWH problems of order $N = 20$. We used the Gomes generator [1] and generated 10 balanced instances for problems with H holes, and did it for a series of different H around the difficulty peak. On each instance, each algorithm had a limited time length t-max of 200 seconds to solve, after that we considered the run as failed.

Comparing to the standard recommended heuristic (msd)

In Figure 4.23, we again show robustness and run time, this time for balanced QWH of order 20. MH(10×3) consistently outperforms min-size domain both in terms of robustness and run time. The graphs show two versions of MH(10×3), one with linear time-limit increase, and one with the order of magnitude increase every 30 restarts. All problems have solutions.

Comparing to randomized-restarts on min domain

Randomized restarts (RR, introduced in Chapter 3) is regarded to be the best method for QWH [51]. We compare MH(10×3) with RR on QWH, in analogy to what we have done on scheduling. Specifically, we implement RR using msd as variable ordering heuristic (with randomized tie-breaking), and using a random value ordering heuristic. For both MH and RR, we again implement the two versions of time-slicing, linear and magnitude, as introduced in Section 4.6.

In Figure 4.24, RR is better than MH almost everywhere, regardless of which time slicing mechanism we use. MH performs slightly better with time slices increased by a magnitude every loop of restarts, for which version we report the statistic on the frequency of failure.

failure frequency [%]		
Size [N]	min size domain	MH magnitude
150	0	0
170	70	20
190	100	50
210	60	20
230	70	0
250	60	0
270	30	0
290	20	0

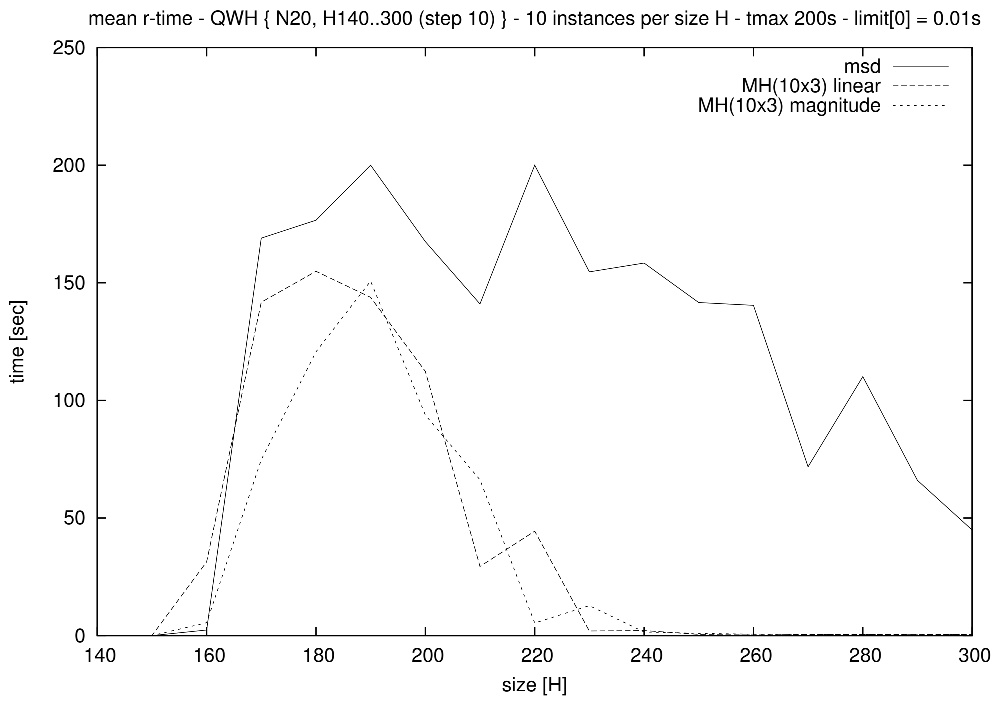


Figure 4.23: Comparing MH against msd on QWH: frequency of failure to solve within *tmax* (top); mean r-time (bottom).

failure frequency [%]		
Size [N]	RR magnitude	MH magnitude
150	0	0
160	0	0
170	30	20
180	40	50
190	20	50
200	10	30
210	0	20
220	10	0

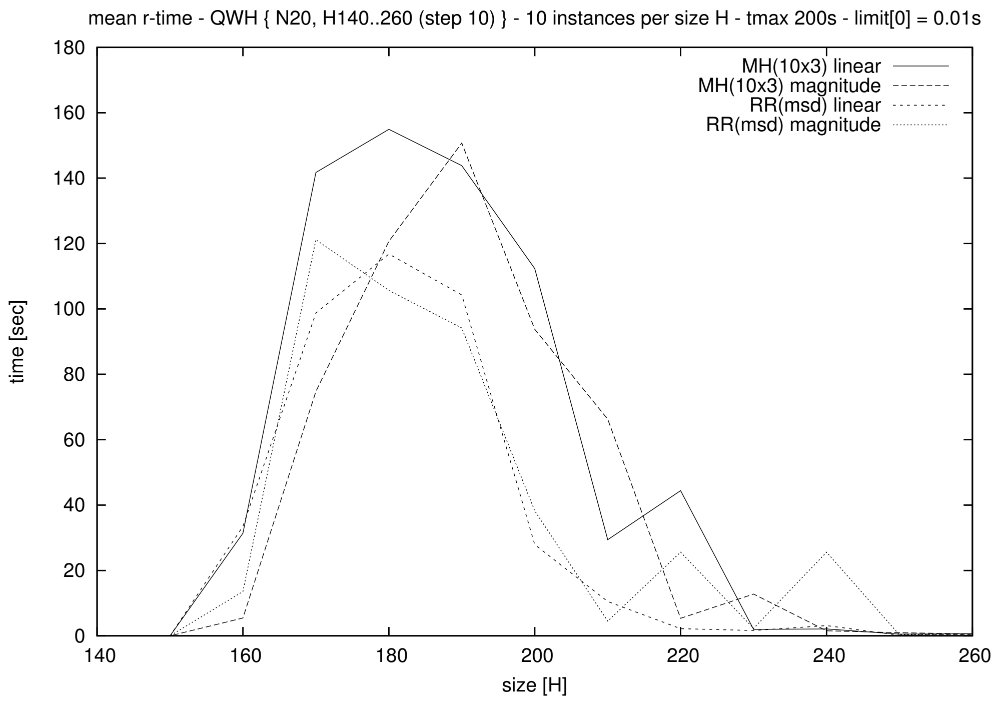


Figure 4.24: Comparing MH against RR on QWH: frequency of failure to solve within *tmax* (top); mean r-time (bottom).

4.6.10 Discussion

Results have shown that MH is clearly better than msd and RR on our scheduling problem class, whose hard instances are located where the solubility is only 10% (Figures 4.19, 4.20, 4.21). On the other hand, MH was still better than msd but poorer than RR on QWH (Figures 4.23, 4.24), whose problem instances are instead all feasible by definition. Thus, the indication is that MH may perform better on those problems which have no solution. For such problems, the solver has to explore the entire search tree - this is always necessary, in order to prove there is no consistent and complete assignment of values to the set of decision variables.

Unlike RR, MH performs a restarting mechanism over a fixed set of variable and value ordering heuristics. Further, the set of heuristics have been designed according to some patterns of the problem. For example, on the scheduling problem we sort the parties (variables) by size and time, and the tables (values) by capacity. Furthermore, for what concerns both the variable and value order selections, MH explores the search space through symmetric paths. In fact, there are heuristics selecting variables in opposite or orthogonal orders - e.g. by increasing rather than by decreasing party size, and breaking ties by increasing rather than by decreasing start time (i.e. using the same or the opposite tie-breaking policy). And there are also heuristics selecting values in opposite orders - i.e. by increasing and by decreasing table capacity.

Note that, the RR version we have used in our tests does not record the *no-good* assignments. In the long run, the randomization process can then end up repeating some search paths which have already been proved unsuccessful in previous restarts. We could compare with an extended RR version which caches the no-good states, so that the correspondent paths are considered only once, or we could also extend MH to store no-goods. However, caching partial assignments requires extra memory and run time. In fact, no-goods are going to be numerous and heavy to store, and the expense may be an issue already for the size of our original problem.

4.7 Chapter summary

We have represented table management as a scheduling problem with fixed start and end times. We have developed a model to represent and solve the problem using single tables (i.e. assuming that tables cannot be joined together). We have designed pre-solving checks and additional constraints, and developed a search approach based on multiple heuristics, aiming to improve search efficiency and robustness. We have tested the final model on two problem classes, restaurant scheduling and quasi-group with holes.

We have shown the benefit of both additional constraints and multiple heuristics search on the original scheduling problem dimensioned upon the Eco restaurant. Results have been confirmed over different distributions of bookings. Specifically, we considered uniform and realistic start times, as well as constant, uniform, and realistic dinner durations. For the original problem we can find a seating plan for a full booking sheet (e.g. 60 parties, or 200 covers) in less than 0.3 seconds on average.

We have tested over a larger scheduling problem, considering an hypothetical restaurant with 100 tables, and finally, over (balanced) quasi-groups with holes of order 20. We have shown that our model is more robust than the standard recommended heuristic, with significantly fewer failures to solve within the time limit. Further, the use of multiple heuristics does not degrade the run time (or efficiency) - in fact, on average it improves the run time. We have also compared to randomized restarts, the leading method for one of our problem classes (QWH) and which uses a similar restart policy. We have shown that the multi heuristic approach is poorer in run time and robustness on QWH, but better on our scheduling problem class.

We can conclude that the additional constraints and the multiple heuristics enhanced our initial model. We can now solve the static decision problem in very reasonable time. We achieved a robust and efficient solution to our scheduling problem, even scaling to sizes larger than the original. Further, MH is also competitive in solving QWH, and can possibly represent an effective method for constraint solving in general.

Chapter 5

Modelling table configurations and seating plan flexibility

5.1 Introduction

In Chapter 4 we developed a constraint model to solve the static decision problem based on scheduling with single (or non-combinable) tables. We now discuss how the model has been extended to tackle many of the real features of restaurant table management. Specifically, the new model represents table configurations, allowing multiple joining and separation of tables in the same evening. Further, we use two different approaches for modelling the flexibility of seating plans to accommodate future meal requests.

The first approach, based on explicit constraints, is important in order to ensure and to maintain seating plans of acceptable quality. We introduce specific constraints to quickly detect (and reject) any booking request whose allocation in the current seating plan would considerably deteriorate the restaurant usage and capacity. For instance, we implement a constraint to avoid large unusable time slots between meals, e.g. a one hour slot between the end of one meal and the start of the next meal in the same table is a waste, if we consider a standard dinner duration of two hours. Similarly, we implement a constraint to avoid seating small parties into over-sized tables, e.g. selling a six seater table to a party of two people is also a waste, if we are expecting more parties of size three or more to arrive.

The second approach, based on optimizing an objective function, is important in order to improve a given seating allocation, say after a change in table usage (e.g. a delay), or after the accommodation of a new booking. The improvement is again based on flexibility, i.e. the aim is to achieve seating plans potentially more flexible to accommodate future meal requests. We describe three flexibility measures and extend them to use some knowledge about the future table demand, based on booking patterns.

The ultimate goal of this chapter is to develop a solution which can provide good quality seating plans in reasonable time. In order to achieve that, we design some implied constraints to speed up the search process with more constraint propagation. Further, we present a study concerning the design and selection of search algorithms for anytime solutions, finalized to produce a more efficient search and a more practical solution to the optimization process.

With respect to our thesis, this chapter shows how constraint programming can be used to model and solve the new, more realistic, static decision problem, and how careful modelling can improve the efficiency. Further, we use simple examples to give first evidence on how constraint programming can be used for flexible and dynamic reconfiguration of tables - to provide good advice on which tables have to be joined and when in order to get a more flexible seating plan. Finally, when we model future knowledge, we provide more examples to motivate how constraint programming can be used to managing uncertainty to build flexible solutions.

5.2 Representing table configurations

The CSP representation for the problem with single tables was discussed in Section 4.3 and Section 4.4. We now extend the model to represent table configurations.

5.2.1 Example

Figure 5.1 shows a problem instance with five parties (top) and a possible allocation (bottom). Note that P_2 is allocated into T_2 and T_3, since the example considers that T_3 can be joined onto T_2 to serve from 4 to 7 people.

Party	**Size**	**Start**	**End**	**Table**
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**
$T_1[2]$	P_1	P_1	P_4	P_4
$T_2[3]$	P_2	P_2	P_5	P_5
$T_3[3]$	P_2	P_2		
$T_4[4]$		P_3	P_3	

Figure 5.1: Problem instance (top); seating plan with table configuration (bottom).

5.2.2 CSP model with table configurations

In our constraint model, the variables are the parties, the possible values represent the tables at which they are seated, and the constraints enforce limits on party size, simultaneous use, etc.. If tables can be combined, then domains and constraints need to be modified, so that it is possible to assign a party to a combined table, while ensuring that another party cannot simultaneously be seated at one of the consituent tables.

Decision variables and possible values

To represent table configurations, we leave the domains unchanged, but we change the constraints - if a party of large size is assigned to conjoined tables i and j, then in the solution we assign the party to table i, and activate a constraint which stops table j being simultaneously assigned to any other party.

The set of decision variables and the set of possible values are:

$$X = \{P_1, P_2, .., P_N\}\ , \quad \text{with } N \text{ number of parties ;}$$
$$Y = \{T_1, T_2, .., T_M\}\ , \quad \text{with } M \text{ number of tables .}$$

Initial domains and constraints

Let J be the set of all groups of tables which can be joined:

$$J = \{(T_{m_1}, .., T_{m_j}) \mid 1 \leq m_1 < .. < m_j \leq M, (T_{m_1}, .., T_{m_j})\ \textit{can join}\}\ .$$

For each group $(T_{m_1}, T_{m_2}, .., T_{m_j}) \in J$, let the minimum party size and the maximum party size for which that group can be used be (respectively):

$$(T_{m_1}, .., T_{m_j}).minPS\ , \quad \text{and} \quad (T_{m_1}, .., T_{m_j}).maxPS\ .$$

In particular, for $j = 1$ we obtain the case of single tables:

$$(T_{m_1}).minPS = 1\ , \quad \text{and} \quad (T_{m_1}).maxPS = T_{m_1}.capacity\ .$$

Note that we consider groups with tables ordered by increasing number. Therefore no group is the permutation of another, i.e. all groups are distinct sets of tables.

For the Eco restaurant, considering the set of tables ordered by the actual table number, there is no group of tables $\{T_{m_1}, T_{m_2}, T_{m_3}\}$, $m_1 < m_2 < m_3$, such that T_{m_1} can be combined either with T_{m_2} or with T_{m_3}. More generally, for many restaurants there exists a numbering of the set of tables (e.g. for Eco is the actual table numbers) such that the following condition is valid.

Condition - For each table T_{m_1}, $m_1 = 1..M$, and for each party size PS, there is at most one group of tables $(T_{m_1}, T_{m_2}, .., T_{m_j}) \in J$, $1 \leq j \leq M$, such that $(T_{m_1}, T_{m_2}, .., T_{m_j}).minPS \leq PS \leq (T_{m_1}, T_{m_2}, .., T_{m_j}).maxPS$.

For example, in Eco, the group of tables (T_{17}, T_{18}, T_{19}) can be combined, $(T_{17}, T_{18}, T_{19}).minPS = 8$, $(T_{17}, T_{18}, T_{19}).maxPS = 11$, and there is no other group of tables starting with table T_{17} that can be used to accommodate any size in the range 8 to 11.

The condition is valid for many restaurants, and it is necessary for our constraint model of table configurations (the description continues next) to be applicable. However, the condition rules out, for example, the case where we have four 2-seater tables (T_A, T_B, T_C, T_D) arranged in a square, and we want to seat 4 people: we could combine T_AT_B or T_BT_C or T_CT_D or T_DT_A, and any numbering scheme is going to violate the condition. Figure 5.2 shows an example with $A = 1$, $B = 2$, $C = 3$, $D = 4$. Note that T_1 can join with either T_2 or T_4 to seat the four people, so the condition does not hold.

In general, for any restaurant for which the graphical representation of the possible table combinations forms a graph with one or more cycles (e.g. as in Figure 5.2), the condition does not hold. Our model would need to be modified to cope with "cyclic" table combinations, although this will have to be the subject of future work.

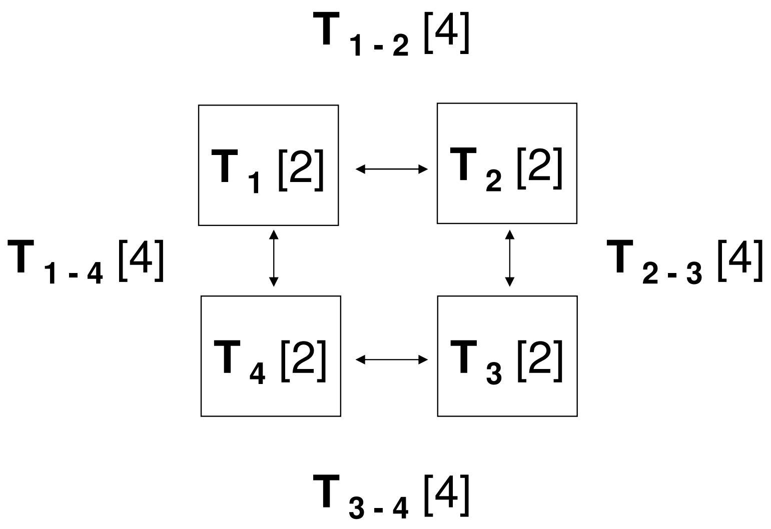


Figure 5.2: Group of four tables of capacity 2, arranged in a square, where each pair of adjacent is combinable and has capacity 4. The graph of possible combinations forms a cycle.

Continuing with our CSP model, to represent the assignment of an entire group of tables $(T_{m_1}, .., T_{m_j}) \in J$ to a party P_n, we simply assign $P_n = T_{m_1}$, and we make all the values in the set (i.e. $T_{m_1}, T_{m_2}, .., T_{m_j}$) unavailable for any other variable correspondent to a party overlapping with P_n. For example, in Figure 5.1, as party P_2 occupies tables T_2 and T_3, we assign the value T_2 to the variable P_2 and we exclude the values T_2 and T_3 from the domains of parties P_1 and P_3.

More formally, we call $T_m.maxGCap$ the maximum capacity of any configurable group of tables where T_m is the first table. The initial domain of each decision variable P_i is then defined as:

$$D_i = \{\ T_j \mid T_j.maxGCap \geq P_i.size\ \}\ .$$

Occupancy constraint - Figure 5.3 shows the procedure which defines the new occupancy constraints $C_{1_{m_a m_b n_1 n_2}}$. Specifically, for any group of configurable tables, $(T_{m_1}, T_{m_2}, .., T_{m_j}) \in J$, and for any party P_{n_1} for which the group is of suitable capacity, $C_{1_{m_a m_b n_1 n_2}}$ ensures the party occupies the group of tables (i.e. the value T_{m_1} is assigned to the variable P_{n_1}) if and only if all tables $\{T_{m_2}, .., T_{m_j}\}$ remain unoccupied (i.e. any other variable P_{n_2} correspondent to a party overlapping with P_{n_1} is not assigned any value in $\{T_{m_2}, .., T_{m_j}\}$).

$$\begin{array}{l}
\forall\ (T_{m_1}, T_{m_2}, .., T_{m_j}) \in J,\ j \geq 2 \\
\forall\ (n_1 = 1..N, n_2 = 1..N),\ n_1 \neq n_2,\ \exists\, t \in R^I\ s.t.\ P_{n_1} \in X_t \wedge P_{n_2} \in X_t \\
(T_{m_1}, T_{m_2}, .., T_{m_j}).minPS\ \leq\ P_{n_1}.size\ \leq\ (T_{m_1}, T_{m_2}, .., T_{m_j}).maxPS \\
\forall\ (m_a = m_1, m_b = m_2..m_j)\ s.t.\ T_{m_b} \in D_{n_2} \\
\\
C_{1_{m_a m_b n_1 n_2}} = (P_{n_1} \neq T_{m_a}) \vee (P_{n_2} \neq T_{m_b})
\end{array}$$

Figure 5.3: Occupancy constraint.

$C_{1_{m_a m_b n_1 n_2}}$ checks all the necessary conditions to avoid overlapping but one, i.e. $P_{n_1} = T_{m_1} \Rightarrow P_{n_2} \neq T_{m_1}$. We could simply include m_1 into the range of m_b in order to ensure that such condition is also verified. However, note that the case is already covered by the original constraint C_{1_t} - which, being a global constraint, is expected to propagate the condition more effectively.

Both constraints C_{1_t} and $C_{2_{khij}}$ maintain the original form:

$$\begin{aligned}
&\forall\, t \,\in\, R^I\,, \\
&C_{1_t} \;= \textit{alldifferent}\{X_t\}\;; \\
\\
&\forall\,(k,h,i,j)\text{ s.t.} \\
&(i,j) \,\in\, (1..N,1..N), i \,< j \\
&(k,h,P_i.size,P_j.size) \,\in\, B\,, \\
&C_{2_{khij}} \;=\; (P_i \neq T_k) \vee (P_j \neq T_h)\;.
\end{aligned}$$

Pre-solving check C_3 is:

$$C_3 = maxNC \leq RC\;,$$

where $maxNC$ is the maximum number of covers at any time t (as defined in Section 4.4) and RC is the capacity of the restaurant, which in the new model becomes the capacity of the restaurant configuration which can seat the largest number of people.

Formally, a restaurant configuration R_{CONF} is a partition of the set of all tables into configurable groups. Then, we can define $R_{CONF} \subseteq J$ as a subset of distinct groups of tables in J such that each table $T_1, T_2, ..., T_M$ appears in one and exactly one group.

The number of people a restaurant configuration can seat is:

$$R_{CONF}.\textit{capacity} = \sum_{(T_{m_1},T_{m_2},..,T_{m_j})\in R_{CONF}} (T_{m_1},T_{m_2},..,T_{m_j}).maxPS\;.$$

Then, RC can be defined as:

$$RC = \max_{R_{CONF}\subseteq J} R_{CONF}.\textit{capacity}\;.$$

Pre-solving check C_{4_i} has the form:

$$\forall\, i \in \{minPS..maxPS\},\ C_{4_i} = maxNP_i \leq M_i\ .$$

where the only difference from the original version is that M_i is now the maximum number of distinct groups of tables which can serve a party of at least i people.

$$M_i = \max_{R_{CONF} \subseteq J} \sum_{(T_{m_1},..,T_{m_j}) \in R_{CONF},\ (T_{m_1},..,T_{m_j}).minPS \leq i \leq (T_{m_1},..,T_{m_j}).maxPS} 1\ .$$

Finally, the symmetry breaker $C_{5_{ij}}$ has the original form:

$$C_{5_{ij}} = P_i < P_j, \forall\, (P_i, P_j)\ \in\ E,\ i < j\ .$$

Example

Figure 5.4 shows the resulting constraint model for the simple problem of Figure 5.1. As we assume tables T_2 and T_3 can be joined, with $(T_2, T_3).minPS = 4$ and $(T_2, T_3).maxPS = 7$, then $T_2.maxGCap = 7$ and $T_3.maxGCap = 3$. There are two possible restaurant configurations, i.e. $R_{CONF_1} = \{(T_1), (T_2), (T_3), (T_4)\}$, and $R_{CONF_2} = \{(T_1), (T_2, T_3), (T_4)\}$. 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. Note that D_2 contains the value T_2, to represent the possibility for P_2 to use the configuration (T_2, T_3). The time range is $R = \{0, 1, 2, 3\}$, while the set of times that are relevant for our constraints is $R^I = \{1, 2\}$. Constraints C_{1_1} (defined in $t = 1$), $C_{1_{2\ 3\ 2\ 1}}$, and $C_{1_{2\ 3\ 2\ 3}}$, ensure that parties P_1, P_2, and P_3 occupy different tables. Constraint C_{1_2} (defined for $t = 2$) is sufficient to ensure that P_3, P_4, and P_5 occupy different tables. As the presence of table configurations has no influence on constraints $C_{2_{khij}}$ and $C_{5_{ij}}$ (discussed in Section 4.4), we assume $B = E = \emptyset$. C_3 is $true$, in fact $maxNP = 9$, which happens at $t = 1$, while $RC = 13$, which is the number of people that can be accommodated using R_{CONF_2}. Finally, C_{4_2}, C_{4_3}, and C_{4_4} are also true, as, in the order, the parties of size at least 2, 3, or 4 are fewer than the groups of tables which can simultaneously serve at least 2, 3, or 4 people (respectively). In particular, note that $M_4 = 2$ is due to R_{CONF_2}, which can simultaneously accommodate two parties of 4, one in configuration (T_2, T_3)

and one in table T_4.

$$\begin{array}{rcl}
X & : & P_1, P_2, P_3, P_4, P_5 \\[1em]
D & : & D_1 = \{T_1, T_2, T_3, T_4\},\ D_2 = \{T_2, T_4\},\ D_3 = \{T_2, T_3, T_4\}, \\
 & & D_4 = \{T_1, T_2, T_3, T_4\},\ D_5 = \{T_1, T_2, T_3, T_4\} \\[1em]
C & : & C_{1_1} = \mathit{alldifferent}(P_1, P_2, P_3) \\
 & & C_{1_{2\,3\,2\,1}} = (P_2 \neq T_2)\ \vee\ (P_1 \neq T_3) \\
 & & C_{1_{2\,3\,2\,3}} = (P_2 \neq T_2)\ \vee\ (P_3 \neq T_3) \\
 & & C_{1_2} = \mathit{alldifferent}(P_3, P_4, P_5) \\
 & & C_3 = maxNC \leq RC, \\
 & & \qquad maxNC = P_1.size + P_2.size + P_3.size = 9,\ RC = 13 \\
 & & C_{4_2} = maxNP_2 \leq M_2,\ maxNP_2 = 3,\ M_2 = 4 \\
 & & C_{4_3} = maxNP_3 \leq M_3,\ maxNP_2 = 2,\ M_2 = 3 \\
 & & C_{4_4} = maxNP_4 \leq M_4,\ maxNP_2 = 1,\ M_2 = 2
\end{array}$$

Figure 5.4: CSP representation for the example of Figure 5.1.

5.3 Modelling Eco with table configurations

We now define the CSP model with table configurations for the restaurant Eco. We focus on the parts which differs from the model with single tables, i.e. initial domains D_i, occupancy constraints $C_{1_{m_a m_b n_1 n_2}}$, and pre-solving checks C_3 and C_{4_i}. X, Y, C_{1_t}, $C_{2_{khij}}$, and $C_{5_{ij}}$ remain unvaried.

5.3.1 Initial domains

Considering the new model with table configurations, the initial domains for the restaurant Eco are now larger. Specifically, depending on the party size, each decision variable has an initial domain as represented in Table 5.1. For clarity, domains have been split between those considering only single tables (second column) and those considering only groups of more than one table (third column).

Table 5.1: Domains in Eco: considering only single tables (second column); and considering only combined tables (third column).

Party size [people]	**Initial domains (only single tables)**	**Initial domains (only combined tables)**
1	$T_1, T_2, .., T_{12}, T_{14}, .., T_{24}$	$\emptyset$
2	$T_1, T_2, .., T_{12}, T_{14}, .., T_{24}$	$\emptyset$
3	$T_1, T_2, T_3, T_6, T_9, T_{10}, T_{11}, T_{14}$ $T_{15}, T_{16}, T_{17}, T_{24}$	$T_{18}, T_{19}, T_{21}, T_{22}$
4	$T_1, T_2, T_6, T_9, T_{10}, T_{11}, T_{14}$ $T_{15}, T_{16}, T_{17}, T_{24}$	$T_{18}, T_{19}, T_{21}, T_{22}$
5	$T_1, T_2, T_6, T_{11}, T_{14}, T_{15}, T_{16}, T_{24}$	$T_{17}, T_{18}, T_{21}, T_{22}$
6	$T_1, T_2, T_6, T_{14}, T_{15}, T_{16}, T_{24}$	T_{17}, T_{18}, T_{21}
7	T_{16}, T_{24}	$T_1, T_5, T_{17}, T_{18}, T_{21}$
8	T_{24}	$T_1, T_5, T_{14}, T_{15}, T_{17}, T_{18}, T_{21}$
9	$\emptyset$	$T_1, T_5, T_{14}, T_{15}, T_{17}, T_{21}$
10	$\emptyset$	$T_1, T_{14}, T_{15}, T_{17}, T_{21}$
11	$\emptyset$	$T_1, T_{14}, T_{15}, T_{17}$
12	$\emptyset$	T_1, T_{14}, T_{17}
13..16	$\emptyset$	T_{14}, T_{17}
17..30	$\emptyset$	T_{14}

The third column shows, for example, how the new model allows 3 more possible solutions to accommodate a party of 6, compared to the model for single tables. In fact, a party of such size can go into tables T_{17}, T_{18} (represented by value T_{17}), or into T_{18}, T_{19}, T_{20} (represented by value T_{18}), or into T_{21}, T_{22}, T_{23} (represented by value T_{21}). The new model can now seat parties of sizes larger than 8, and up to 30. In total, as discussed in Section 2.3, there are 16 possible table configurations which can be combined to allow 386 different restaurant layouts. Obviously, inefficient table allocations are not included in the combined table allocations, e.g. we do not seat a party of size 1 at a combined table.

5.3.2 Occupancy constraints

The number of occupancy constraints depends on the booking sheet. For example, for any party P_{n_1} of 6 people, and for any party P_{n_2} overlapping in time with P_{n_1}

we have the five constraints:

$$\begin{aligned}
C_{1_{17\ 18\ n_1\ n_2}} &= (P_{n_1} \neq T_{17}) \ \vee \ (P_{n_2} \neq T_{18}) \ , \\
C_{1_{18\ 19\ n_1\ n_2}} &= (P_{n_1} \neq T_{18}) \ \vee \ (P_{n_2} \neq T_{19}) \ , \\
C_{1_{18\ 20\ n_1\ n_2}} &= (P_{n_1} \neq T_{18}) \ \vee \ (P_{n_2} \neq T_{20}) \ , \\
C_{1_{21\ 22\ n_1\ n_2}} &= (P_{n_1} \neq T_{21}) \ \vee \ (P_{n_2} \neq T_{22}) \ , \\
C_{1_{21\ 23\ n_1\ n_2}} &= (P_{n_1} \neq T_{21}) \ \vee \ (P_{n_2} \neq T_{23}) \ .
\end{aligned}$$

Recall that the *alldifferent* constraint C_{1_t} covers the case $(P_{n_1} \neq T_m) \vee (P_{n_2} \neq T_m)$, with $m = 17, 18, 21$.

5.3.3 Restaurant capacity and pre-solving checks

The capacity of the restaurant Eco using single tables is 83. If we allow table configurations the maximum capacity achievable is 96, and the correspondent restaurant configuration is:

$$\begin{aligned}
R_{CONF} \ = \{ \ & (T_1, T_{14}), T_2, T_3, T_4, (T_5, T_6), T_7, T_8, T_9, T_{10}, T_{11} \\
& T_{12}, (T_{15}, T_{16}), (T_{17}, T_{18}, T_{19}, T_{20}), (T_{21}, T_{22}, T_{23}), T_{24} \ \} \ .
\end{aligned}$$

Therefore, C_3 becomes:

$$C_3 \ = maxNC \leq RC \ (RC = 96) \ .$$

Table 5.2 combines the second and third columns of Table 5.1, to show the number of parties Eco can simultaneously accommodate for each party size. The second last column reports the sum of the possible ways to accommodate using either single tables or table configurations. The last column counts only the number of distinct tables and configurations, i.e. tables and configurations which can be adopted simultaneously. For example, Eco has 7 tables plus 3 table configurations which can serve 6 people, for a total of 10 possible ways of accommodation. However, 2 of the 3 table configurations, (T_{17}, T_{18}) and (T_{18}, T_{19}, T_{20}), cannot be adopted simultaneously. Therefore, the maximum number of parties of size 6 eating at the same time is 9. We use the last column to implement our second pre-solving check, i.e.:

$$C_{4_s} = maxNP_s \leq M_s \quad \forall\ s = 1..30\ .$$

Table 5.2: Number of tables and configurations in Eco, categorized by party size.

Party size $[s]$	**single tables**	**table configurations**	**total**	**distinct** $[M_s]$
1	23	0	23	23
2	23	0	23	23
3	12	4	16	14
4	11	4	15	13
5	8	4	12	10
6	7	3	10	9
7	2	5	7	6
8	1	7	8	6
9	0	6	6	5
10	0	5	5	4
11	0	4	4	3
12	0	3	3	2
13..16	0	2	2	2
17..30	0	1	1	1

5.4 Additional constraints

With the introduction of table configurations the problem becomes harder. Testing the current model on problem instances dimensioned upon the restaurant Eco we found that most instances were unacceptably long to solve. Therefore we designed two new constraints, aiming to improve search efficiency: a capacity constraint, and a new symmetry breaker.

5.4.1 Capacity constraint

The procedure to generate the new capacity constraint C_{6_t} is shown in Figure 5.5. The constraint ensures that, at any time $t \in R^I$, the number of usable tables UTN_t

is not less than the number of parties NP_t. The upper bound number of usable tables is initialized to M minus NP_t (number of parties present at time t), as each of the parties is going to use at least 1 table. That number is then decreased by $(j-1)$ each time j tables are joined, to represent the extra $j-1$ tables which are used. Note that, unlike C_{4_i} which is a simple check performed before starting the search, C_{6_t} is a real constraint, i.e. it propagates during search.

```
$\forall\, t \in R^I$
        $UTN_t =$                 new constrained variable (Nb. usable tables)
        $UTN_t.domain =$          $\{-M, .., -1, 0, 1, .., M\}$
        $UTN_t.max =$             $M - NP_t\ ,\quad (NP_t\ Nb.\ parties\ in\ X_t)$

            $\forall$             $i = 1..N,\ P_i \in X_t$
                                  $j = 2..M,\ (T_{m_1}, T_{m_2}, .., T_{m_j}) \in J$
                                  $(T_{m_1}, .., T_{m_j}).minPS \leq P_i.size \leq (T_{m_1}, .., T_{m_j}).maxPS$

                                  $UTN_t.max- = (j-1) \times (P_i == T_{m_1})$

        $C_{6_t}\ = UNT_t \geq 0$
```

Figure 5.5: Procedure to generate the new capacity constraint.

Figure 5.6 shows a feasible problem instance with five overlapping parties (top) and a partial allocation during search involving the first four and performed without constraint C_{6_t} (bottom). The example considers that T_2 (T_4) can be joined onto T_1 (T_3) to serve 3 people. We assume that search allocated parties P_1, P_2, P_3, and P_4 one by one in the order. Note that this search path has now reached a dead end because there is no room for the next party P_5 in the partial seating plan. The infeasibility of the current search path can be spotted much earlier using constraint C_{6_t}. In fact, right after the search allocates P_1 into T_1 and T_2, C_{6_t} would forbid P_2 from being allocated into T_3 and T_4, as this would give the inconsistency $C_{6_t} : 6 - 5 - (2-1) - (2-1) \ngeq 0$. This example shows how C_{6_t} can save the search from going deeper along the wrong subtree.

5.4.2 A symmetry breaker on identical tables

Figure 5.7 represents a symmetry breaker constraint $C_{7_{m_a\ m_b}}$ on identical tables, by which we mean those tables of same capacity and which cannot be joined with

Party	**Size**	**Start**	**End**	**Table**
P_1	3	0	2	?
P_2	3	1	3	?
P_3	2	1	3	?
P_4	2	0	2	?
P_5	2	1	3	?

Table[size]	**0**	**1**	**2**	**3**
$T_1[2]$	P_1	P_1		
$T_2[2]$	P_1	P_1		
$T_3[2]$		P_2	P_2	
$T_4[2]$		P_2	P_2	
$T_5[3]$		P_3	P_3	
$T_6[3]$	P_4	P_4		

Figure 5.6: Problem instance (top); partial allocation during search (bottom).

others. The constraint posts an order among any two identical tables. For instance, considering the allocation of any two identical tables, if we exclude any particular preference, there is no difference for both the restaurant and the customer point of view between allocating more parties in one rather than in the other table. Therefore, for any pair of such tables $(T_{m_a},\ T_{m_b})$, $m_a < m_b$, we only allow allocations where the number of parties in T_{m_a} (NP_{m_a}) is greater or equal than the number of parties in T_{m_b} (NP_{m_b}).

```
$\forall m \in 1..M$
        $NP_m =$ new constrained variable (Nb. parties in $T_m$)
        $NP_m.domain = \{0, 1, .., N\}$
        $\forall i = 1..N,\quad NP_m + = (P_i == T_m)$

$\forall m_a = 1..M,\ m_b = 1..M,\ m_b > m_a$
                $T_{m_a},\ T_{m_b}$ singles
                $T_{m_a}.capacity = T_{m_b}.capacity$

                $C_{7\ m_a\ m_b} = NP_{m_a} \geq NP_{m_b}$
```

Figure 5.7: Procedure to generate the symmetry breaker for single tables of equal capacity.

Figure 5.8 shows two symmetrical seating plans for the same set of parties. Note again that, if we exclude any particular preference, there is no difference for either the restaurant or the customer between the allocations. With constraint $C_{7_{1\ 2}}$ defined using the procedure of Figure 5.7, only the allocation displayed at the top would be considered by the search process. $C_{7_{m_a\ m_b}}$ prunes the search space, forbidding symmetrical allocations on single tables of equal capacity like T_1 and T_2. In particular, this avoids repeating symmetrical mistakes during search - i.e. exploring wrong symmetrical paths of the search tree.

Table[size]	**0**	**1**	**2**	**3**
$T_1[4]$	$P_1[3]$	$P_1[3]$	$P_3[4]$	$P_3[4]$
$T_2[4]$		$P_2[3]$	$P_2[3]$	

Table[size]	**0**	**1**	**2**	**3**
$T_1[4]$		$P_2[3]$	$P_2[3]$	
$T_2[4]$	$P_1[3]$	$P_1[3]$	$P_3[4]$	$P_3[4]$

Figure 5.8: Symmetrical allocations: (top) accepted; (bottom) rejected.

In the restaurant Eco there is a group of 5 identical tables of capacity 2, i.e. T_4, T_5, T_7, T_8, and T_{12}, and a group of 2 identical tables of capacity 4, i.e. T_9 and T_{10} (described in Section 2.3). The correspondent set of constraints is described in Figure 5.9.

$$\begin{aligned}
C_{7\ 4\ 5} &= NP_4 \geq NP_5 \\
C_{7\ 5\ 7} &= NP_5 \geq NP_7 \\
C_{7\ 7\ 8} &= NP_7 \geq NP_8 \\
C_{7\ 8\ 12} &= NP_8 \geq NP_{12} \\
C_{7\ 9\ 10} &= NP_9 \geq NP_{10}
\end{aligned}$$

Figure 5.9: Symmetry breaker for identical tables in Eco.

5.4.3 Evaluating the effect of the new constraints

To evaluate the effect of the new constraints, we performed an experiment similar to those discussed in Section 4.6 for the model with single tables. Specifically, we considered the set of 23 tables of the restaurant Eco, and generated random booking sheets. The size of the booking sheets ranges from 5 to 100 parties in steps of 5, for a total of 20 different sizes. Start times are uniformly distributed from 4 p.m. to 10 p.m. over 25 time units of 15 minutes. We also consider a uniform (4 to 8 time units) distribution of dinner durations, and a distribution of party sizes in the range [1..8] (with average 3.84, as estimated for the restaurant Eco). We perform tests over 100 instances for each possible size of booking sheet, for a total of 2000 problems. We adopted the search algorithm based on MH(11×3), *magnitude* version. We impose the time-limit for each run to 17 seconds (as in the experiments of Section 4.6.7), after that the problem is regarded as unsolved.

Figure 5.10 compares the model using the original constraints, CSP(X, D, C_1, C_2, C_3, C_4, C_5), to the same model extended with the two additional constraints, C_6 and C_7. We consider three extensions, one with C_6, one with C_7, and one with both. The table at the top of the figure displays the number of instances that were still unsolved after the time limit, while the graph at the bottom shows the mean running time to solve. Further, in the graph, the curve from top left to bottom right represents the solvability, i.e. the percentage of instances which are feasible. As the number of parties per booking sheet increases, the solvability decreases. In particular, all instances with fewer than 35 parties have solution, all those with more than 75 parties allow no solution, while about 50% of instances with 60 parties are feasible.

Considering the version without the additional constraints, the frequency of failure is unacceptable, with up to 52% of instances still unsolved after 17 seconds (Figure 5.10 top). Further, we can see that the corresponding run-time increases significantly as the problem grows in size, with a peak at 70 parties. After that, the booking sheets become too crowded, and so more instances are likely to be infeasible because they exceed the restaurant capacity. In such cases, our pre-solving checks on the maximum numbers of people (C_3) and parties (C_4) are violated, so the problems are found infeasible, and the solution process can be stopped without performing any search. This explains the decrease in both run-time and failures after 70 parties - more details on the benefit of C_3 and C_4 were presented in Chapter 4.

The model extended with the new capacity constraint (C_6) performs significantly better, with the curve of mean run-time showing a peak of only 0.63 seconds at 65 parties, and with all instances solved within the time limit.

Symmetry constraints reduce the size of the search tree, so, in general, they make the search process more efficient (Chapter 3). The result concerning our model extended with the symmetry breaker on identical tables (C_7) was quite unexpected. Even though we have only two small sets of identical tables (Figure 5.9), i.e. we were not demanding any large improvement, we can observe how, for the model with C_7, both run time and failure frequency are actually slightly worse compared to the results concerning the model with the original constraints. This suggests that the effort in terms of time for generating and propagating C_7 might be greater than the time we save by searching over a solution space of smaller size, i.e. the one reduced by C_7.

A more speculative explanation is that the symmetry breaker may work against our multiple heuristic procedure. In fact, one benefit of MH is that it explores the search space through multiple searches, each following a distinct direction. Combined with C_7, which forces each search to follow only a subset of possible directions, this synergy among the set of heuristics may become less effective.

Finally, the version including both C_6 and C_7 has a slightly worse performance than the case with only C_6, which confirms that the real benefit comes from C_6. Our model for the next part of the dissertation will be based on C_1, C_2, C_3, C_4, C_5, and C_6.

5.5 Constraints on poor table assignments

Common heuristic guidelines utilized by restaurants are: (i) avoid allocations with long dead-zones; and (ii) avoid allocations with over-sized tables. To ensure solutions of acceptable quality, we designed new constraints based on these two guidelines.

failure frequency [%]				
Size [N]	C_1,..,C_5	C_1,..,C_5,C_6	C_1,..,C_5,C_7	C_1,..,C_5,C_6,C_7
10	0	0	0	0
20	0	0	0	0
30	0	0	0	0
40	0	0	0	0
50	4	0	4	0
55	10	0	10	0
60	21	0	23	0
70	52	0	54	1
80	28	0	28	0
90	8	0	8	0
100	1	0	1	0

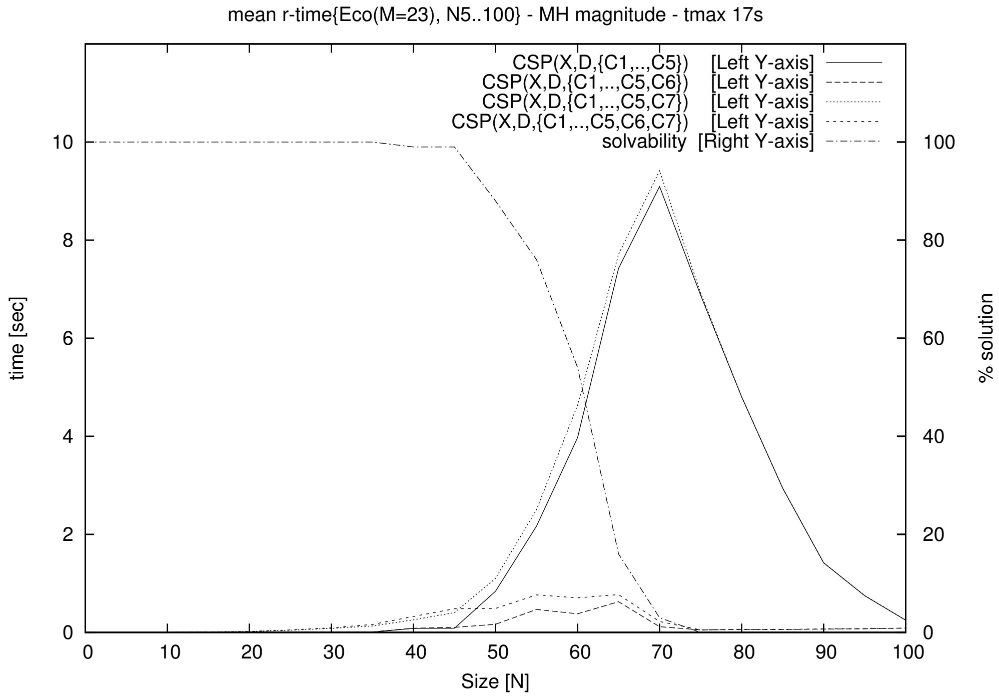


Figure 5.10: Evaluating the additional constraints: frequency of failure to solve within *tmax* (top); mean r-time (bottom). The basic version of CSP includes constraints C_1 to C_5 (Chapter 4). We compare it against versions extended with C_6 (capacity constraint), with C_7 (symmetry breaker), or with both C_6 and C_7. In the graph, the curve from top-left to bottom-right (solvability) is the percentage of instances with a feasible solution. Each point in the graph is an average over 100 instances.

5.5.1 Long dead-zones constraint

If there is a party with a reservation from 5 to 7, and a second party with a reservation from 8 to 10, then the booker does not allocate the two parties into the same table. Otherwise, from 7 to 8 the table is going to be idle and unusable to serve other parties. Idle and unusable times (dead-zones) can be short, i.e. 15-30 minutes, or long, i.e. more than 30 minutes but less than the standard dinner duration (typically 2 hours). For bookings, a dead-zone of 15-30 minutes cannot be judged a priori - it may reveal to be either a waste in table usage, if the first dinner finishes on time, or a useful slack time to absorb delays, in case the first dinner is late. Dead-zones longer that 30 minutes are instead very likely to significantly deteriorate the reservation potential of a dinner session - so they are generally avoided, unless the manager has a strong preference for a particular customer. A formal representation of the long dead-zone constraint, $C_{8_{n_a\ n_b}}$, is shown in Figure 5.11 (top). The constraint ensures that no pair of parties allocated into the same table generates a dead-zone between 30 and 120 minutes.

5.5.2 Over-sized tables constraint

Generally, unless it is Valentine's Day, or unless the booker has a strong preference for a particular customer, a booking for 2 people is always allocated into a table of capacity two or (at most) three. Otherwise, using a 4-seater or larger table the restaurant would waste a significant amount of reservation potential. A formal representation of the over-sized tables constraint, $C_{9_{n\ m}}$, is shown in Figure 5.11 (bottom). The constraint ensures that no party of size 1 or 2 is allocated into a 4-seater or larger table.

5.5.3 Efficiency with the constraints on poor table assignments

To evaluate the effect of constraints C_8 and C_9, we repeated the same type of experiment previously used for the evaluation of constraints C_6 and C_7. Specifically, we now compare the model CSP(X, D, C_1, C_2, C_3, C_4, C_5, C_6), to the same model extended with: (i) C_8; (ii) C_9; and (iii) both C_8 and C_9. Figure 5.12 shows the results, again, in terms of failures to solve within the time limit (top), and the

$$\begin{aligned}
\forall \quad & n_a = 1..N,\ n_b = 1..N,\ n_a \neq n_b,\ \ 30' < (P_{n_b}.start\ -\ P_{n_a}.end)\ < 120' \\
& j = 1..M,\ (T_{m_1}, T_{m_2}, .., T_{m_j}) \in J \\
& (T_{m_1}, .., T_{m_j}).minPS\ \leq\ P_{n_a}.size\ \leq\ (T_{m_1}, .., T_{m_j}).maxPS \\
& \\
& C_{8_{m_1\ ..\ m_j\ n_a\ n_b}}\ =\ (P_{n_a} \neq T_{m_1}) \vee (P_{n_b} \notin \{T_{m_1}, T_{m_2}, .., T_{m_j}\})
\end{aligned}$$

$$\begin{aligned}
\forall \quad & n = 1..N,\ P_n.size \in \{1, 2\},\ \ m = 1..M,\ T_m.capacity \geq 4 \\
& \\
& C_{9_{n\ m}}\ =\ (P_n \neq T_m)
\end{aligned}$$

Figure 5.11: Procedures to generate: (top) the constraint on long dead-zones in the range 30 to 120 minutes; (bottom) the constraint on over-sized tables for parties of size 1 and 2.

mean run-time to solve (bottom). We can see that the version without C_8 and C_9 has the best run-time and failure frequency. When we include the constraints on long dead-zones the performance gets worse for problems of size up to 50. In particular, for problems with 50 parties we have a peak of 2.08 seconds in the graph, and 3% of failures to solve before the time limit. Instead, if we include the constraint on over-sized tables there is a peak of run-time of 5.52 seconds in correspondence of problems with size 55, with a considerable increase in the number of failures (reaching 30% at 55 parties). Finally, considering the version that includes both C_8 and C_9, for problems of size up to 50 both run-time and failure frequency show a further increase compared to the previous version with C_9, while the performance is the same after 50 parties.

Adopting the restrictions on poor table allocations is important in order to build booking sheets and seating plans that guarantee the potential to accommodate an acceptable number of covers - an initial set of parties “poorly” allocated can irreversibly reduce the capacity of the restaurant. Instead, once the restaurant is getting close to full occupancy, and we are expecting only a few more table requests, any extra party is a bonus for the restaurant. For example, if the restaurant has only one four-seater table left, and gets only one more request of size two, the best solution would be to accept the party, and allocate it to the over-sized table. In general, when booking sheets and seating plans are crowded, and we are getting close to the reservation target, restaurants can gain extra flexibility by ignoring any restriction on dead-zone and over-sized table allocations.

The solution we adopt hybridizes the model without restrictions on poor table allocations, and the model with both the constraints on long dead-zone and over-sized table allocations. For our case study, the Eco restaurant has a reservation target of around 185 covers, and an average party size of 3.84, which means that the target is reached after 45-50 parties. Thus, for problems of size up to 40 we use CSP(X, D, C_1, .., C_6, C_8, C_9), i.e. we include both the restrictions C_8 and C_9, while for problems with more that 40 parties we use the model CSP(X, D, C_1, .., C_6), i.e. we ignore the restrictions. If we consider Figure 5.12, the hybrid model would coincide with the upper line for $N \leq 40$, and with the lower line for $N > 40$. The hybrid model would be efficient over the entire range of problem sizes, with a peak of run-time of 1.34 seconds, and a peak of failure frequency of 5%, both at 40 parties. Further, the performance would be significantly better for smaller problem sizes (see curve concerning the model with C_8 and C_9), and especially for larger problem sizes (curve concerning the model without C_8 and C_9).

Finally, Figure 5.13 shows how the introduction of constraints C_8 and (especially) C_9 reduces the number of instances with a solution. This again supports our decision to use the restrictions on poor table allocations only until the restaurant has reached a certain level of occupancy - for the hybrid model, after 40 parties the solvability curve would shift from the one concerning CSP(X, D, C_1, .., C_6, C_8, C_9) to that concerning CSP(X, D, C_1, .., C_6), with a considerable increase in the number of solutions that becomes acceptable, i.e. in flexibility to accept the final table requests (and the final extra covers).

5.6 Flexibility and optimization

In Chapter 4, we described a satisfaction problem: i.e. the model does not consider optimization, 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

failure frequency [%]				
Size [N]	C_1,..,C_6	C_1,..,C_6,C_8	C_1,..,C_6,C_9	C_1,..,C_6,C_8,C_9
10	0	0	0	0
20	0	0	0	0
30	0	0	2	2
40	0	0	5	5
50	0	3	8	13
55	0	0	30	30
60	0	0	27	27
70	0	0	3	3
80	0	0	0	0
90	0	0	0	0
100	0	0	0	0

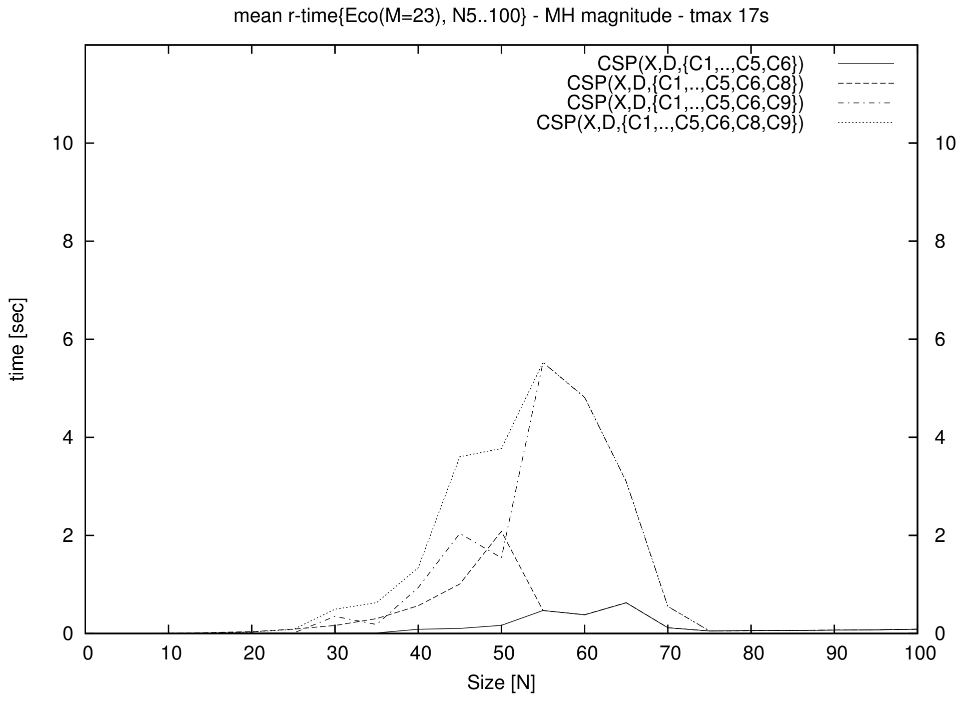


Figure 5.12: Evaluating the constraints on poor table allocations: frequency of failure to solve within *tmax* (top); mean r-time (bottom). We compare the version of CSP including constraints C_1 to C_6 against versions extended with C_8 (dead zone constraint), with C_9 (oversized table constraint), or with both C_8 and C_9.

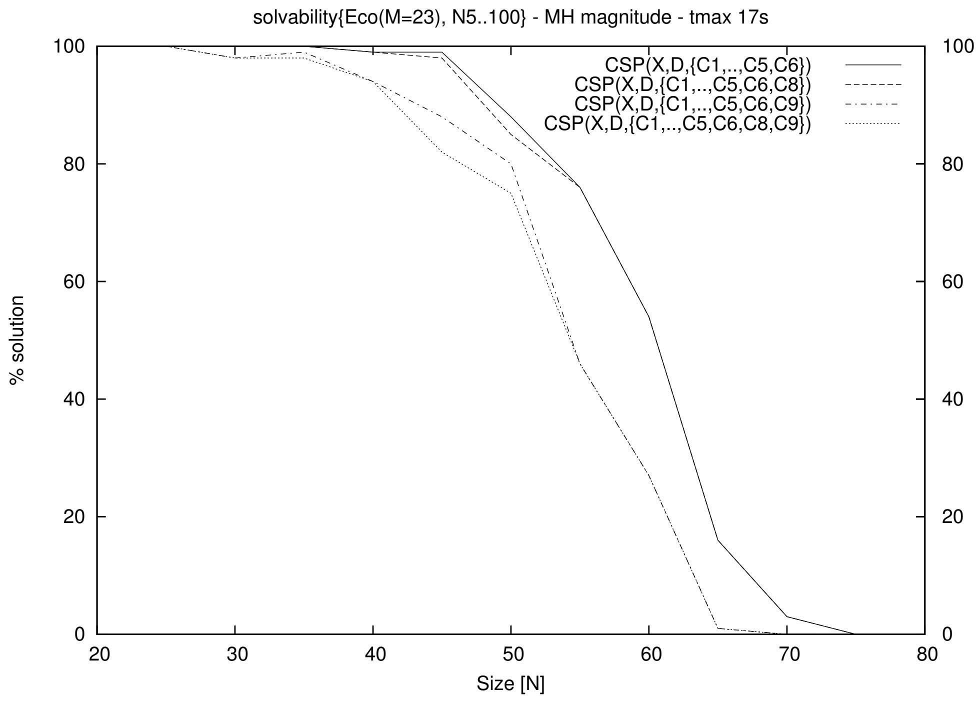


Figure 5.13: Graph showing how the introduction of the constraints on poor table allocations, C_8 and C_9, affects the solvability curve.

ability to accept future bookings. In this section, we describe some measures to estimate the quality of a solution, and the algorithms which use the measures 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 maximizing the covers. Thus, after each change, we should be searching for:

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

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:

$$\text{argmax}_{seating\ plan}[\text{flexibility}]\ .$$

For each problem instance, we then perform an anytime branch-and-bound search, optimizing for flexibility.

5.6.1 Constraint implementation of flexibility

In this section, we extend the constraint model to represent a measure of flexibility. Specifically, we map a seating allocation (or plan) to a grid of distances, and then use the map to measure the flexibility of the allocation. Later in the section, we will introduce three different measures of flexibility.

Grid of distances

Figure 5.14 shows an example of seating plan allocating three parties (top), and the corresponding grid of distances G (bottom), for a restaurant with 3 tables and time horizon 9. The value assigned to each square $G[i][t]$ in the grid indicates the number of squares (or *distance*) from time t before table T_i becomes unavailable. In particular, $G[i][t] = 0$ indicates that table T_i is occupied at time t.

Tables	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
T_1	P_1	P_1	P_1						
T_2	P_1	P_1	P_1				P_3	P_3	P_3
T_3		P_2	P_2	P_2					

G	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	0	0	0	6	5	4	3	2	1
2	0	0	0	3	2	1	0	0	0
3	1	0	0	0	5	4	3	2	1

Figure 5.14: Example of seating plan with three parties (top); and corresponding grid of distances (bottom).

Computing G

Let M be the number of tables, and T be the time horizon discretized in T time units. Given a seating plan, we superimpose a grid G of size $M \times T$ over the plan. Each square is a constrained variable whose initial domain ranges between 0 and $T - i$, with i column of the grid where the square is located. $T - i$ represents the *distance* to the last time unit. Then, the procedure to generate the map is represented in Figure 5.15. Specifically, we first consider each party P_n in turn, and, in

$$\begin{array}{ll} \forall & n \in 1..N,\ j = 1..M,\ (T_{m_1}, .., T_{m_j}) \in J \\ & (T_{m_1}, .., T_{m_j}).minPS \ \leq \ P_n.size \ \leq \ (T_{m_1}, .., T_{m_j}).maxPS \\ & C_{MAP_a} : P_n = T_{m_1} \rightarrow G[m_1, .., m_j][P_n.start, .., P_n.end] = 0 \\ & \\ \forall & m \in 1..M,\ t \in 2..T, \\ & C_{MAP_b} : G[m][t] - G[m][t-1] \leq 1 \end{array}$$

Figure 5.15: Procedure for mapping seating allocations into grids of distances.

turn, we consider each table (or group of tables) $(T_{m_1}, .., T_{m_j})$ in the set of possible J. For each pair $\langle P_n, (T_{m_1}, .., T_{m_j})\rangle$ such that the capacity of $(T_{m_1}, .., T_{m_j})$ is suitable for the size of P_n, C_{MAP_a} ensures that if the party is accommodated into $(T_{m_1}, .., T_{m_j})$ (i.e. the variable P_n is assigned the value T_{m_1}), all the grid squares corresponding to tables T_{m_1}, T_{m_2}, .., T_{m_j}, over the columns from P_n.start to P_n.end, are given a value 0, to represent the occupancy.

The second constraint C_{MAP_b} is necessary to work out, for each slot (table, time) in the grid, the number of time units available before the table becomes occupied (or unavailable). For each table m and time t, C_{MAP_b} ensures that the difference between the variable $G[m][t]$ and the precedent variable $G[m][t-1]$ is at most 1. Doing this, for example, an occupancy of table m at time 4, mapped to $G[m][4] = 0$ by C_{MAP_a}, would change the domains of time 1, 2, and 3, from $G[m][1] = [0..9]$, $G[m][2] = [0..8]$, and $G[m][3] = [0..7]$ to $G[m][1] = [0..3]$, $G[m][2] = [0..2]$, and $G[m][3] = [0..1]$. In general, after C_{MAP_b}, the maximum value of each square on each table represents the (horizontal) distance from the square to the first unavailable square on the same table.

In Figure 5.16, the grid at the top represents the initial domains of the squares for an empty seating plan. During search, as we know, tables are assigned to parties according to some ordering heuristics. After each assignment, constraints C_{MAP_a} and C_{MAP_b} propagate over G. The second grid shows the effect of constraint C_{MAP_a} on the domains corresponding to the occupied slots - each square in grey indicates that a party is occupying the table at that time unit. The third grid shows the effect of constraint C_{MAP_b}. The maximum value of each slot (*table*, *time*) now represents the number of squares (or *distance*) before the table becomes unavailable.

Finally, after the search reaches a solution we perform the following step:

$$G[m][t].min = G[m][t].max \ , \quad \forall \quad m = 1..M, \ t = 1..T \ .$$

The last grid in Figure 5.16 shows the final version of the allocation map, which assigns the exact distance value to each cell in G.

Flexibility measures

We developed a first measure of flexibility based on the number of *usable start times* for future requests:

$$flexibility_{\,US} = |US| \ , \quad US = \{(m,t) : G[m,t] \geq d\} \ ,$$

Table	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
T_1	0..9	0..8	0..7	0..6	0..5	0..4	0..3	0..2	0..1
T_2	0..9	0..8	0..7	0..6	0..5	0..4	0..3	0..2	0..1
T_3	0..9	0..8	0..7	0..6	0..5	0..4	0..3	0..2	0..1

Table	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
T_1	0	0	0	0..6	0..5	0..4	0..3	0..2	0..1
T_2	0	0	0	0..6	0..5	0..4	0	0	0
T_3	0..8	0	0	0	0..5	0..4	0..3	0..2	0..1

Table	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
T_1	0	0	0	0..6	0..5	0..4	0..3	0..2	0..1
T_2	0	0	0	0..3	0..2	0..1	0	0	0
T_3	0..1	0	0	0	0..5	0..4	0..3	0..2	0..1

Table	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
T_1	0	0	0	6	5	4	3	2	1
T_2	0	0	0	3	2	1	0	0	0
T_3	1	0	0	0	5	4	3	2	1

Figure 5.16: Example of seating map: initial domains (top); mapping the occupancy (second top); rearranging the domains of distances (second bottom); final map, assigning the exact distance to the domain of each cell (bottom).

$$flexibility_{\,WUS} = \sum_{(m,t)\in US} T_m.capacity\ .$$

Considering the map of distances G, squares with numbers less than a standard dinner duration d are ignored, as they do not represent usable start times. $T_m.capacity$ is the capacity of table T_m and represents the weight of our measure.

In the restaurant Eco neither booker nor floor manager perform any specific optimization. However, they base their decisions on a (vague) policy, which can be interpreted as another measure of flexibility. The aim of the policy is to maintain a seating plan which minimizes *dead-zones* - where, for each table, a dead-zone can be defined as the time length, between the occupancies of consecutive parties, which is not sufficiently long for accommodating any future request. We then developed a second measure of flexibility as follows:

$$flexibility_{DZ} = -|DZ| \;, \quad DZ = \{(m,t) : 0 < G[m,t] < d\} \;,$$

$$flexibility_{WDZ} = - \sum_{(m,t) \in DZ} T_m.capacity \;.$$

Now, squares with numbers more than a standard dinner duration d or squares which are not between two consecutive parties are ignored, as they are not dead-zones.

To evaluate the quality of a seating plan, another measure of flexibility can be designed looking at the number of seatings (or parties) each table can potentially serve. For example, in the restaurant Eco the aim is to achieve three seatings per table at the end of the night. To count the number of seatings, we consider only those squares in G with values equal to whole number multiples of a standard dinner duration d, and ignore the remaining cells. We compute the flexibility based on seatings as follows:

$$flexibility_{NS} = |NS| \;, \quad NS = \{(m,t) : \exists k \in \mathbb{N}^+, G[m,t] = k \times d\} \;,$$

$$flexibility_{WNS} = \sum_{(m,t) \in NS} T_m.capacity \;.$$

As an illustration, Figure 5.17 shows a restaurant with 2 tables, T_1 of capacity 3 and T_2 of capacity 7, and the map of 2 possible seating plans for 3 parties: P_1 (size 3, start 6, end 9), P_2 (size 7, start 2, end 5), P_3 (size 2, start 2, end 5). The evening is divided into 8 time units (i.e. $T = 8$), and we assume the standard dinner duration is $d = 3$. The grid cells show the number of time units available. Then, considering the three flexibility measures, we obtain the following estimates: flexibility$_{WUS}$ (top) $= (2 \times 3) = 6$; flexibility$_{WUS}$ (bottom) $= (2 \times 7) = 14$; flexibility$_{WDZ}$ (top) $= -(1 \times 7) = -7$; flexibility$_{WDZ}$ (bottom) $= -(1 \times 3) = -3$; flexibility$_{WNS}$ (top) $= (1 \times 3) = 3$; and flexibility$_{WNS}$ (bottom) $= (1 \times 7) = 7$. Thus, all the flexibility measures prefer the second seating plan. Note that, the corresponding versions with no weights on table capacity would not distinguish between the two plans: flexibility$_{US}$ (top) = flexibility$_{US}$ (bottom) = 2; flexibility$_{DZ}$ (top) = flexibility$_{DZ}$ (bottom) = 1; and flexibility$_{NS}$

(top) = flexibility$_{NS}$ (bottom) = 1.

Table[size]	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
$T_1[3]$	1	P_3	P_3	P_3	4	3	2	1
$T_2[7]$	1	P_2	P_2	P_2	1	P_1	P_1	P_1

Table[size]	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
$T_1[3]$	1	P_3	P_3	P_3	1	P_1	P_1	P_1
$T_2[7]$	1	P_2	P_2	P_2	4	3	2	1

Figure 5.17: Flexibility map for two possible allocations.

5.6.2 Using future knowledge

Depending on the distribution of future requests, by size and time, the preference over different possible seating plans can change.

Example

Figure 5.18 shows a restaurant with three tables, T_1 of size 2, T_2 of size 2, T_3 of size 3, and two possible seating plans for 4 parties, P_1 (size 2, start 1, end 4), P_2 (size 3, start 3, end 6), P_3 (size 2, start 6, end 9), P_4 (size 2, start 1, end 4). We assume no group of tables can be joined. If we know we are likely to receive a request of size 3 for time 6, then we should prefer the first plan. If we are more likely to receive two requests, both of size 2 and for time 4 or 5, then we should prefer the second plan.

Flexibility with future knowledge

The three measures of flexibility, introduced in the previous section, have been extended to reason about future knowledge. Specifically, if P_{mt} is the probability associated to a request of size $T_m.capacity$ for a dinner starting at time t, the new

Table[size]	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
$T_1[2]$	P_1	P_1	P_1	2	1	P_3	P_3	P_3
$T_2[2]$	P_4	P_4	P_4	5	4	3	2	1
$T_3[3]$	2	1	P_2	P_2	P_2	3	2	1

Table[size]	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
$T_1[2]$	P_1	P_1	P_1	5	4	3	2	1
$T_2[2]$	P_4	P_4	P_4	5	4	3	2	1
$T_3[3]$	2	1	P_2	P_2	P_2	P_3	P_3	P_3

Figure 5.18: Flexibility map for two possible allocations.

measures are:

$$flex_{\ WUS} = \sum_{(m,t)\in US} T_m.capacity \times P_{mt}\ ,$$

$$flex_{\ WDZ} = -\sum_{(m,t)\in DZ} T_m.capacity \times P_{mt}\ ,$$

$$flex_{\ WNS} = \sum_{(m,t)\in NS} T_m.capacity \times P_{mt}\ .$$

Back to the example of Figure 5.18, considering all requests equally likely, e.g. $P_{mt} = 1$ for all $m = 1..M$ and $t = 1..T$, using our measure based on usable start times we would obtain flex$_{WUS}$ (top) $= 3 \times 2 + 1 \times 3 = 9$, and flex$_{WUS}$ (bottom) $= 3 \times 2 + 3 \times 2 = 12$, i.e. the preferred plan is the one at the bottom. Assuming $P_{3\ 6} = 1$ (i.e. we are sure to receive a request for 3 people for time 6), $P_{2\ 4} = 0.5$, $P_{2\ 6} = 0.5$, and $P_{m\ t} = 0$ for any other pair (m, t), we obtain instead: flex$_{WUS}$ (top) $= 2 \times 2 \times 0.5 + 1 \times 3 \times 1 = 5$, and flex$_{WUS}$ (bottom) $= 4 \times 2 \times 0.5 = 4$, i.e. the preferred plan becomes the one at the top.

Booking patterns

The data model we are going to use is based on the expected number of booking requests per party size and time. Specifically, we consider B booking sheets from the past, and count the number of occurrences of each type of booking, by party size and start time. For each $m = 1..M$, $t = 1..T$, our estimate for P_{mt} is then:

$$P_{mt} = \frac{1}{B} \sum_{b=1..B,\ n_b=1..N_b,\ P_{n_b}.size=T_m.capacity,\ P_{n_b}.start=t} 1\ .$$

The pattern of requests varies, e.g. depending on the season, on the occasion (Christmas, Valentine's Day, etc), or on the day of the week, we can have different distributions of parties by size and time. The most evident case concerns Valentine's Day, where most of the parties are couples. Our model should then compute a different P_{mt} for each category (i.e. season, special occasion, day of the week, etc). For example, it should use booking sheets of past Valentine's Days to generate the estimates of table demand for the next Valentine's Day.

Getting the data is often difficult - many restaurants do not take reservations, and others do not store the booking sheets. Further, even though past booking sheets often represent the best data available for estimating table demand, they keep a record only of the accepted requests, so we do not know the number and nature of those others that could not be satisfied. More accurate estimates for the terms P_{mt} should be based on all the requests, accepted and rejected. In order to do that, booking systems should be changed so that the complete information on all the requests can be retrieved.

Flexibility and booking patterns for Eco

As a first attempt at using estimates of future requests, we implemented a simplified version of P_{mt}, weighting only the distributions by time (and not by party size). P_t is then the probability estimate of a request of any size for time t:

$$P_t = \frac{1}{B} \sum_{b=1..B,\ n_b=1..N_b,\ P_{n_b}.start=t} 1\ .$$

Thus, we obtain:

$$flex_{\ WUS} = \sum_{(m,t)\in US} T_m.capacity \times P_t\ ,$$

$$flex_{WDZ} = -\sum_{(m,t)\in DZ} T_m.capacity \times P_t \ ,$$

$$flex_{WNS} = \sum_{(m,t)\in NS} T_m.capacity \times P_t \ .$$

We considered 21 booking sheets (three weeks) provided by the restaurant Eco. We divided them into two categories, (i) Sundays, and (ii) any other days of the week. We computed the estimates P_t, scaling the frequency of requests over time down to weights in the range 0 to 4, and obtaining the distributions of Table 5.3. For example (second and second last columns), considering booking requests for a Sunday, we can read that the frequency of 4 o'clock bookings is four times the frequency of booking requests for 9:30 p.m., while, for another day, we can expect two bookings for 9:30 p.m. for every booking for 4 o'clock. For simplicity, here we only report the values by steps of 30 minutes, i.e. for even time units 0, 2, .., 24. Note that, the peak of table load (or demand) on Sundays is between 4 and 7 o'clock, while on another day it ranges between 6 and 9 o'clock.

Table 5.3: Booking distribution in Eco, for Sundays or for any other day.

clock	400	430	500	530	600	630	700	730	800	830	900	930	1000
time-unit	0	2	4	6	8	10	12	14	16	18	20	22	24
Sundays	4	4	4	4	4	4	4	3	2	2	1	1	0
other days	1	1	2	2	4	4	4	4	4	4	4	2	1

5.7 Anytime algorithm

In this section, we combine the constraint model with an anytime algorithm based on our measures of flexibility. The aim is to provide reasonably flexible solutions in acceptable time. The basic algorithm is represented in Figure 5.19.

The procedure takes as input the (new) CSP model, generated from the restaurant description and from the set of bookings, and an initial solution (or seating plan). The new CSP model contains G (with domains D_G, and constraints $C_{MAP_{a,b}}$), and the constrained variable *flexibility* (as implemented in Section 5.5).

```
INPUT:      CSP = $(X, D, C_1, C_2, .., C_9) \cup (G, D_G, C_{MAP_{a,b}}$, flexibility$)$
            currentSolution = initialSolution

$C_{FLEX}$   =  flexibility $>$ currentSolution.flex

while       Solver.improve$(CSP \cup C_{FLEX}, SA)$ == true
            currentSolution = Solver.getSolution()
            $C_{FLEX}$ = flexibility $>$ currentSolution.flex

return      currentSolution
```

Figure 5.19: Basic algorithm for anytime solutions.

Then we use the flexibility measure of the initial solution as lower bound for the next solution in the improvement process - we impose a constraint C_{FLEX} on the constrained variable *flexibility*. The flexibility value *currentSolution.flex* is computed using the map of distances (e.g. Figure 5.16). In the *while* loop, *Solver.improve(.)* considers the constraint model CSP and starts the solution process, propagating the satisfaction, mapping, and flexibility constraints over the search tree, and adopting a search algorithm SA (see next). The search looks for a new seating plan of increased flexibility. If a better plan is found the current solution is updated, and so is the constraint on the flexibility lower bound. The loop is repeated until no further improvement is possible, i.e. assuming no time limit, until we reach an optimal seating plan. Note that the current (best) solution is available at any time during the solution process.

5.7.1 Selecting the search algorithm

We designed four versions of search algorithms for anytime solutions, i.e.:

SA_1	=	MH based, *search state reset before new improvement search* ,
SA_2	=	MH based, *search state resumed before new improvement search* ,
SA_3	=	SH based, *search state reset before new improvement search* ,
SA_4	=	SH based, *search state resumed at new improvement search* .

Versions SA_1 and SA_2 considers a search based on our multiple heuristic algorithm (i.e. MH *magnitude*, discussed in Chapter 4). Version SA_3 and SA_4 consider instead a search based on a single ordering heuristic (i.e. SH, composed by *min-size-domain* as variable ordering, and by *min-table-capacity* as value ordering). For SA_1 and SA_3, the search restarts from the root of the search tree at each loop of improvement. SA_2 and SA_4, instead, continue from the state left in the last loop (i.e. from the leaf of the search tree corresponding to the current solution).

5.7.2 Objective

Apart from selecting the algorithm with the best anytime performance, the current objective is to show that, given a set of parties and an initial seating plan, we can provide, in reasonable time, new plans with a significant increase in flexibility - according to a certain measure of flexibility. Note that the measures we designed in Section 5.5 represent only estimates of flexibility, and therefore, after each improvement loop, we cannot say whether the potential to accommodate future parties has really increased. This matter, i.e. assessing the accuracy of the flexibility measures, will be discussed in Chapter 6, where we simulate and solve the dynamic problem.

5.7.3 Experiments

As just discussed, selecting one rather than another flexibility measure is not so significant for the purpose of the current tests. Therefore, here we use the measure based on usable start times, setting *flexibility* = *flexibility*$_{WUS}$. Further, we assume a uniform distribution of future requests, i.e. we set $P_t = 1$ for any time t.

We considered a restaurant description representing the Eco restaurant, i.e. the 23 tables of capacity ranging from 2 to 8, and the 16 possible table configurations (described in Chapter 2). We tested the 3 versions of anytime algorithms over a set of problem instances. Each instance is a list of parties, each with a size, a fixed start time, and a fixed duration. We generated instances of size 1 to 50 parties, one instance for each size. Further, in the order, party size, start time, and duration are uniformly distributed in the ranges 1 to 8, 0 to 24 (which we correspond to the range 4:00 p.m. to 10:00 p.m. discretized in 15 minutes units), and 5 to 10 (corresponding to the range 1:15 hours to 2:30 hours). Note that, with an average of 4.5 people per party, a set of 50 parties gives an average of 225 covers, which would require the full capacity of the restaurant Eco.

For each set of parties, the initial plans are pre-computed by solving the satisfaction problem - for doing this we used our original solver based on the multiple heuristic algorithm (discussed in Chapter 4). Then we run each anytime algorithm for 5 minutes (300 sec) over each instance, recording the flexibility value of the variable *currentSolution* at fixed intervals of time, i.e. at the seconds 1, 2, 3, 4, 5, 10, 15, 45, 60, 300.

5.7.4 Results

Figure 5.20 shows the performance profile of the four algorithms, i.e. the flexibility improvement over time (from 1 second to 5 minutes, in log scale) with respect to the flexibility of the initial seating plan. The graphs are the average over all the problem instances, i.e. for all sizes from 1 to 50 parties.

We can see that in the first 3 seconds the two versions based on a single ordering heuristic ($SA_{3,\ 4}$) are slightly better than the versions using our multiple heuristic algorithm ($SA_{1,\ 2}$). As an explanation, in the initial phase of the improvement process we expect that improvements are easier to find, therefore is may be better to use the single (and recommended) ordering heuristic adopted by $SA_{3,\ 4}$.

On the other end, as we found in our experiments in Chapter 4, MH appears to outperform the other methods when we are asked to explore the complete search space, i.e. when problems are mostly infeasible (as in that case), or when we have to perform a deeper or full optimization (as we do here). In fact, back to Figure 5.20, if we can allow more time for the improvement process SA_1 and SA_2 outperform both SA_3 and SA_4. The benefit gets more significant after 10 seconds ($\sim +5\%$ compared to the other versions), and the overall improvement on the initial solution after 5 minutes is around 9%. Note how SA_1 gets slightly better than SA_2 between 5 and 11 seconds, but then the two performances converge again, and after 30 seconds they overlap.

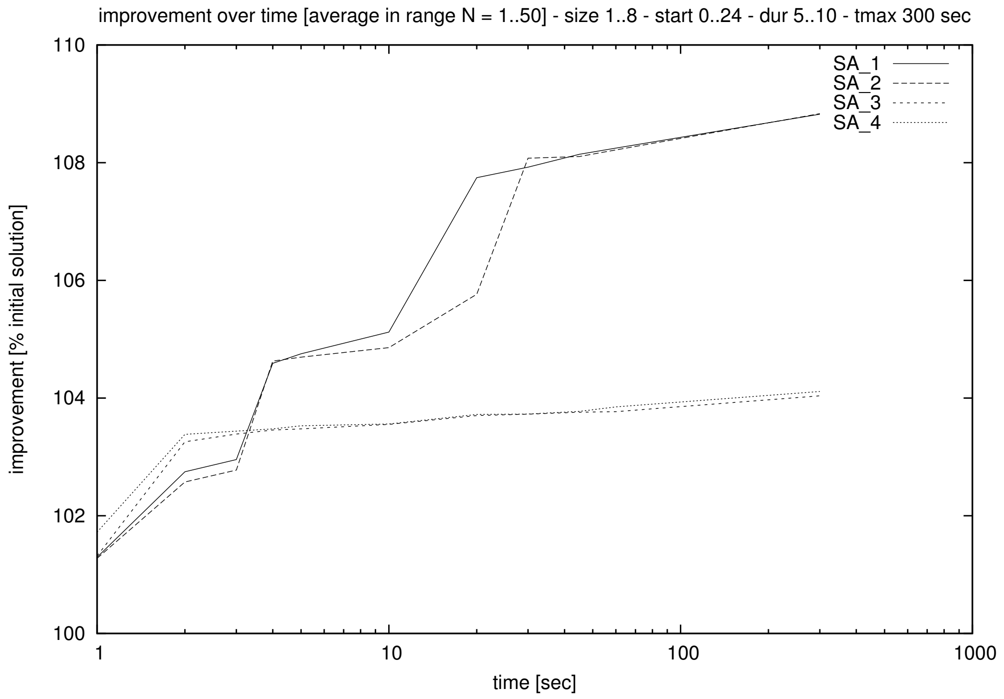


Figure 5.20: Anytime algorithm, performance profile of versions SA_1, SA_2, SA_3, SA_4, with instances of size 1 to 50 parties. Horizontal axis in logarithmic scale.

Finally, version SA_4 is slightly better on average than SA_3. This indicates that, for a search based on a single heuristic it is advisable to start each improvement search from the state reached in the previous loop. The difference, though, is not significant.

Figure 5.20 is an average over instances of size in the range 1 to 50. In general, instances with a very few parties or instances regarding a very crowded booking sheet are the easiest to solve, whether the problem is of satisfaction or of optimization. The results presented in Figure 5.21 are an average over the subset of instances of size 21 to 40, i.e. they focus on the harder region. In this case, SA_1 and SA_2 are always better than SA_3 and SA_4. The final improvement is again around 9%, compared to the other two algorithms, whose performance is poorer in this region ($\sim 2.5\%$ of improvement over the first solution, compared to $\sim 4\%$ of improvement obtained averaging over all the problem sizes 1 to 50). This again confirms that for harder problems a search based on MH is more efficient. Finally, considering the two versions based on MH ($SA_{1,\ 2}$), we can observe that the version restarting the search after each improvement (SA_1) shows an overall slightly better performance compared to the version resuming the search after each improvement (SA_2).

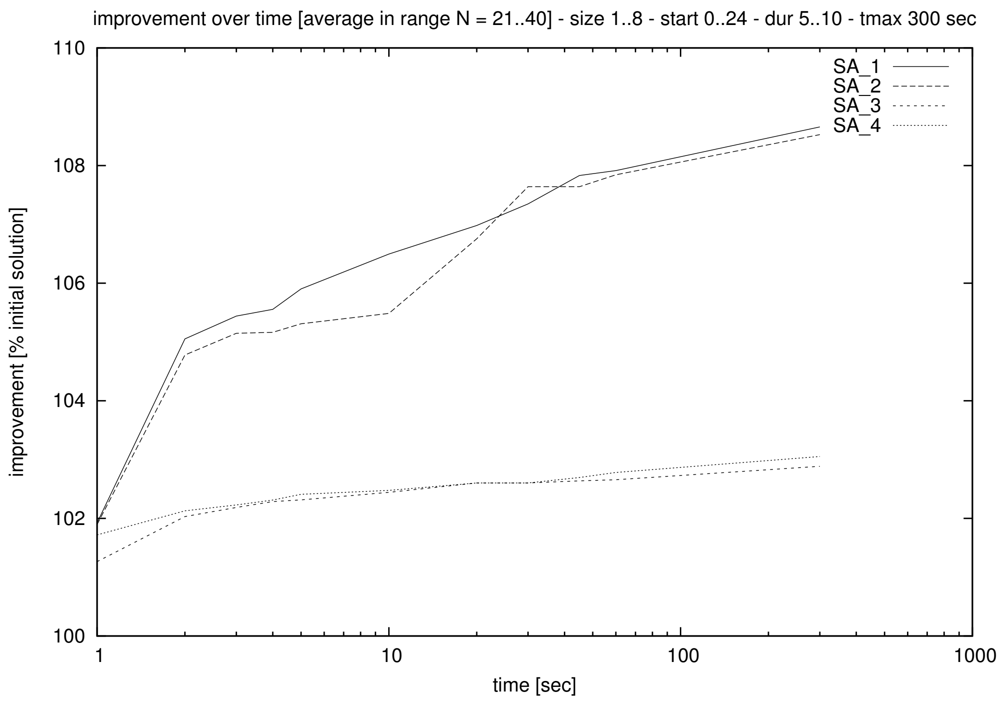


Figure 5.21: Anytime algorithm, performance profile of versions SA_1, SA_2, SA_3, SA_4, with instances of size 21 to 40 parties. Horizontal axis in logarithmic scale.

5.8 Chapter summary

In this chapter we modified the basic restaurant scheduling problem with single tables to include table configurations and seating plan optimization.

We introduced new constraints to represent table configurations, extending the original model, and discussing the new representation for the specific case of the restaurant Eco. Testing our initial model we realized that the new problem involving table configurations was practically intractable for the size of the restaurant

Eco. We therefore designed new constraints to prune the search space. In particular, adding a new constraint on table capacity we achieved a very efficient solution.

We also introduced constraints to control poor table allocations, i.e.: (i) a constraint to avoid long dead-zones between consecutive allocations; and (ii) a constraint to forbid allocations of twos into four seater or larger tables.

We then presented our model for optimization, which is based on measuring seating plan flexibility. Flexibility has also been expressed using constraints. Specifically, we map party allocations into a grid of distances, and express flexibility using the distance of each grid slot (*table*, *time*) to the next unavailable slot on that table. Given the standard dinner duration d (e.g. 2 hours), we proposed three versions of flexibility based on counting: (i) usable start times, i.e. distances longer than d; (ii) dead-zones, i.e. distances shorter than d regarding slots located between consecutive parties; and (iii) number of seatings, i.e. distances multiple of d. The three measures have been extended to weight the distances by the expectation of future table demand over time. In particular, we presented an example of expectation for the case Eco, where weights are based on booking patterns retrieved from past booking sheets.

The constraint model was then combined with an anytime optimization algorithm. Specifically, we designed and compared four versions of algorithms, two using our multiple heuristic search (MH), and two based on a single heuristic. We tested the four versions over problem instances ranging from almost empty to full booking sheets. For each instance, we analyzed the anytime profile of the improvement achieved by each algorithm over an initial seating plan. The overall best performance concerns the versions using MH, which shows an improvement of $\sim 5\%$ after 10 seconds, and of $\sim 9\%$ after 5 minutes, with results regarding an average over all the instances. Excluding from the average the instances concerning almost empty or very crowded booking sheets, which can be easy to solve, the versions based on MH were again outstanding, with an improvement of more than 6% in 10 seconds, and of $\sim 9\%$ in 5 minutes. The versions based on a single heuristic search showed a poorer profile everywhere, apart from the case concerning the interval 0 to 10 seconds and including all (i.e. both easy and hard) instances. The two versions based on MH showed similar performances, with the version restarting the search after each improvement slightly better than the version resuming the search after each improvement.

In conclusion, after careful modelling we have achieved an efficient representation for the new problem with table configurations, and an optimization model with a good anytime profile, i.e. with a reasonable improvement/time ratio.

Chapter 6

Solving the dynamic problem

6.1 Introduction

In Chapter 4, we developed a constraint model to solve a static decision problem based on scheduling with single tables. In Chapter 5, we extended the model to solve a static optimization problem based on scheduling with reconfigurable tables. While in both the previous chapters we tested our techniques on static problems, the aim of this chapter is to show how the same methods can cope with the dynamic problem.

Section 6.2 introduces the dynamic problem, characterizing the process of booking and floor management over time. In Section 6.3, we present our allocation methods, and implement also two versions simulating traditional methods currently adopted in restaurants. Section 6.4 describes an algorithm for solution (or seating plan) stability, which constrains the number of changes during the first phase of search. In Section 6.5, we present the experiments and discuss the results. Specifically, we evaluate our solutions over booking simulations, comparing to the traditional booking policies, and comparing versions with and without optimization. To represent the situation where customers do not have a preference for a specific booking time, we perform new experiments allowing diner's start time flexibility. We evaluate the different flexibility measures described in Chapter 5, including our model of future knowledge. We also test our models on the floor management phase, evaluating flexibility and robustness in managing uncertainty on table demand (walk-ins) and on delays (dinners lasting longer than expected, and late arrivals). Finally, Section 6.6 reports a summary of the chapter.

The results we are going to present show how the CP based model we have achieved represents an effective solution for restaurant table management. For instance, based on booking and floor simulations, and assuming dinners cannot be delayed to accommodate unexpected changes (e.g. to fit extra parties), we show how our solution significantly outperforms the traditional allocation method. The improvement over current systems comes from:

- not committing any table to any party at reservation time, so our solution is more robust (or flexible) for the accommodation of future changes, and in particular for managing the uncertainty on table demand and on delays;

- performing optimization, i.e. we can provide more flexible assignments of parties to tables (in both booking allocation and floor allocation);

- exploiting diners' start time flexibility (i.e. when customers have no specific preference for the time to consume their dinner), again, to preserve more flexible seating plans;

- enhancing uncertainty management, by including some reasoning on future knowledge into our flexibility estimates.

We will show how all these points contribute to a significant increase in the number of customers the restaurant can accommodate.

6.2 The dynamic problem

As described in Chapter 2, restaurant table management is a dynamic problem with two distinct phases: booking and floor management. In both phases, frequent and uncertain changes (e.g. new booking requests, delays, etc.) occur over time, and instant decisions have to be taken in order to accommodate the changes.

6.2.1 Possible changes during the booking phase

The booking phase precedes the dinner session, and the main changes during booking can be:

c_1 new booking requests;

c_2 booking changes (i.e. customers phoning up, asking to change the size or the time of a previous reservation);

c_3 cancellations.

The booking phase starts with an empty booking sheet, where the booker is going to allocate table slots to each (accepted) booking request. When a request (c_1) cannot be accommodated on the current booking sheet, either he can persuade the customer to accept another time, or the request must be declined. Similarly, when there is a booking change in size or time (c_2), he checks whether the booking sheet can accommodate the change, and if not the request gets declined. Finally, when there is a cancellation (c_3), he simply erases the booking from the booking sheet. Note that c_2 can be regarded as a combination of c_3 followed by c_1, i.e. we can first cancel the original booking and then input a new booking request with the new size/time details. The booker needs to be reasonably quick to find out whether or not a booking request can be accommodated (e.g. the customer cannot be held for one hour on the phone). In practice, the booker spends around 10 to 20 seconds to process each request.

6.2.2 Possible changes during floor management

In floor management, changes happen during dinner time, and most of them have immediate effect on the current state of the restaurant. In this phase, c_1, c_2, and c_3 are still possible changes, along with:

c_4 walk-ins (parties arriving without reservation);

c_5 no-shows (parties with reservation who did not turn up);

c_6 parties arriving later or earlier than expected;

c_7 dinners lasting longer or shorter than expected;

c_8 parties turning up with more or fewer people than the table size that was booked;

c_9 parties that arrive believing a booking has been made when none has been recorded;

c_{10} table preferences or parties who do not like the (pre)assigned table.

In floor management, the evening starts with a partially completed booking sheet. The customers have been given definite times, and the aim is to modify the seating plan when changes happen. Walk-ins (c_4) and preferences (c_{10}), similarly to booking requests (c_1) or booking changes (c_2), can be accepted or declined depending on the table availability on the current seating plan. No-shows (c_5) are managed similarly to cancellations (c_3), i.e. reservations expire if the customer does not turn up within a time limit. In Eco, for example, the floor manager assumes that parties with reservation can arrive late by up to 30 minutes, after that he deletes the reservation. For the remaining types of changes (c_5, c_6, c_7, c_8, c_9), it may be infeasible to accommodate them without delaying any party in the current seating plan. The manager has to adapt the current plan to accommodate the change - which usually means that some future parties has to be reallocated or delayed. Similarly to the booker in the booking phase, the floor manager needs to respond quickly to accommodate changes as they happen over time - in practice, 10 to 20 seconds is again a reasonable time to process each change.

6.2.3 Modelling the booking phase using dynamic scheduling

We now present the dynamic problem concerning the booking phase, showing how we solve it, and in particular how the CSP model varies over time - we translate the changes from the restaurant description (i.e. c_1, c_2, c_3) to the CSP model description.

Figure 6.1 shows an example of dynamic problem for the booking phase. On the left we represent the sequence of static instances (each containing the change that generated it), and on the right the corresponding seating plans. The problem starts with an empty list of bookings (top-left), and an empty seating plan (top-right). Then we have the following sequence of 7 changes: new request (P_1, accepted); new request (P_2, accepted); new request (P_3, accepted); booking change ($P_2.size$, accepted); new request (P_4, accepted); cancellation (P_1); new request (P_5, declined, no suitable table available).

Table 6.1 shows how our CSP model changes according to the sequence represented in Figure 6.1. For each change CH_n ($n = 1..7$), the corresponding model is CSP^n = (X^n, D^n, C^n). We can observe how: new bookings (c_1) introduce new variables and domains, and augment the number and the scope of constraints; booking changes (c_2) require changes in the constraint description; and cancellations (c_3) remove variables and domains, and reduce the number and the scope of the constraints.

6.2.4 Modelling the floor phase using dynamic scheduling

We now present the dynamic problem concerning the floor phase. Again, we show how we solve the problem, and how the CSP model varies as changes happen - now changes can be of type c_1, c_2, .., c_9.

Figure 6.2 shows an example of dynamic problem for the floor phase. On the left there is the sequence of static instances, and on the right the seating plans. The problem starts with a partial list of bookings (top-left) and an initial seating plan (top-right), provided by the preceding booking phase (Figure 6.1). The time line t during the floor phase (here $t = 0, 1, 2, 3, 4$) is represented by the double vertical line on the seating plans. Note that allocations become fixed from the moment parties get seated, so when a change happens only the future dinners can be reallocated in order to solve the new problem instance. In this example there are 6 changes: walk-in (time 0, P_6, accepted); walk-in (time 1, P_7, declined, no suitable table available); late arrival (time 1, $P_3.start$, $P_3.end$, accepted); late finish (time 2, $P_6.end$, accepted); no-show (time 2, P_4); arrival with change in size (time 2, $P_3.size$, accepted).

Table 6.2 shows how the CSP model varies according to the dynamics represented in Figure 6.2. For each change CH_n ($n = 1..6$), the corresponding model is CSP$'^n$ = (X'^n, D'^n, C'^n). We can observe how: walk-ins (c_4) introduce new

Party	**Size**	**Start**	**End**	**Table**

P_1	2	0	3	?

P_1	2	0	2	?
P_2	2	0	2	?

P_1	2	0	2	?
P_2	2	0	2	?
P_3	3	1	3	?

P_1	2	0	2	?
P_2	4	0	2	?
P_3	3	1	3	?

P_1	2	0	2	?
P_2	4	0	2	?
P_3	3	1	3	?
P_4	2	2	4	?

P_2	4	0	2	?
P_3	3	1	3	?
P_4	2	2	4	?

P_2	4	0	2	?
P_3	3	1	3	?
P_4	2	2	4	?
P_5	4	0	2	?

Table[size]	**0**	**1**	**2**	**3**
$T_1[2]$				
$T_2[3]$				
$T_3[3]$				
$T_4[4]$				

$T_1[2]$	P_1	P_1		
$T_2[3]$				
$T_3[3]$				
$T_4[4]$				

$T_1[2]$	P_1	P_1		
$T_2[3]$	P_2	P_2		
$T_3[3]$				
$T_4[4]$				

$T_1[2]$	P_1	P_1		
$T_2[3]$	P_2	P_2		
$T_3[3]$		P_3	P_3	
$T_4[4]$				

$T_1[2]$	P_1	P_1		
$T_2[3]$	P_2	P_2		
$T_3[3]$	P_2	P_2		
$T_4[4]$		P_3	P_3	

$T_1[2]$	P_1	P_1	P_4	P_4
$T_2[3]$	P_2	P_2		
$T_3[3]$	P_2	P_2		
$T_4[4]$		P_3	P_3	

$T_1[2]$			P_4	P_4
$T_2[3]$	P_2	P_2		
$T_3[3]$	P_2	P_2		
$T_4[4]$		P_3	P_3	

$T_1[2]$			P_4	P_4
$T_2[3]$	P_2	P_2		
$T_3[3]$	P_2	P_2		
$T_4[4]$		P_3	P_3	

Figure 6.1: Example of dynamic problem on the booking phase: sequence of static instances (left); possible seating plans (right).

Table 6.1: CSP representation for the example of Figure 6.1. Note: (RC, M_2, M_3, M_4) = (12, 4, 3, 2).

i	X^i	D^i	C^i
1	P_1	$D_1 = \{T_1, T_2, T_3, T_4\}$	$C_3 = maxNC \leq RC,\ maxNC = P_1.size = 2$ $C_{4_2} = maxNP \leq M_2, maxNP_2 = 1$ $C_{9_{1\ 4}} = (P_1 \neq T_4)$
2	P_1 P_2	$D_1 = \{T_1, T_2, T_3, T_4\}$ $D_2 = \{T_1, T_2, T_3, T_4\}$	$C_{1_1} = alldiff(P_1, P_2)$ $C_3 = maxNC \leq RC,$ $maxNC = P_1.size + P_2.size = 4$ $C_{4_2} = maxNP_2 \leq M_2,\ maxNP_2 = 2,\ M_2 = 4$ $C_{5_{1\ 2}} = P_1 < P_2,\ C_{9_{1\ 4}} = (P_1 \neq T_4),\ C_{9_{2\ 4}} = (P_2 \neq T_4)$
3	P_1 P_2 P_3	$D_1 = \{T_1, T_2, T_3, T_4\}$ $D_2 = \{T_1, T_2, T_3, T_4\}$ $D_3 = \{T_2, T_3, T_4\}$	$C_{1_1} = alldiff(P_1, P_2, P_3)$ $C_3 = maxNC \leq RC,$ $maxNC = P_1.size + P_2.size + P_3.size = 7$ $C_{4_2} = maxNP_2 \leq M_2,\ maxNP_2 = 3$ $C_{4_3} = maxNP_3 \leq M_3,\ maxNP_3 = 1$ $C_{5_{1\ 2}} = P_1 < P_2,\ C_{9_{1\ 4}} = (P_1 \neq T_4),\ C_{9_{2\ 4}} = (P_2 \neq T_4)$
4	P_1 P_2 (size=4) P_3	$D_1 = \{T_1, T_2, T_3, T_4\}$ $D_2 = \{T_2, T_4\}$ $D_3 = \{T_2, T_3, T_4\}$	$C_{1_1} = alldiff(P_1, P_2, P_3)$ $C_{1_{2\ 3\ 2\ 1}} = (P_2 \neq T_2) \vee (P_1 \neq T_3)$ $C_{1_{2\ 3\ 2\ 3}} = (P_2 \neq T_2) \vee (P_3 \neq T_3)$ $C_{2_{3\ 4\ 3\ 2}} = (P_3 \neq T_3) \vee (P_2 \neq T_4)$ $C_3 = maxNC \leq RC,$ $maxNC = P_1.size + P_2.size + P_3.size = 9$ $C_{4_2} = maxNP_2 \leq M_2,\ maxNP_2 = 3$ $C_{4_3} = maxNP_3 \leq M_3,\ maxNP_3 = 2$ $C_{4_4} = maxNP_4 \leq M_4,\ maxNP_4 = 1$ $C_{6_1} = 4 - 3 - (P_2 == T_2) \geq 0,\ C_{9_{1\ 4}} = (P_1 \neq T_4)$
5	P_1 P_2 P_3 P_4	$D_1 = \{T_1, T_2, T_3, T_4\}$ $D_2 = \{T_2, T_4\}$ $D_3 = \{T_2, T_3, T_4\}$ $D_4 = \{T_1, T_2, T_3, T_4\}$	$C_{1_1} = alldiff(P_1, P_2, P_3),\ C_{1_2} = alldiff(P_3, P_4)$ $C_{1_{2\ 3\ 2\ 1}} = (P_2 \neq T_2) \vee (P_1 \neq T_3)$ $C_{1_{2\ 3\ 2\ 3}} = (P_2 \neq T_2) \vee (P_3 \neq T_3)$ $C_{2_{3\ 4\ 3\ 2}} = (P_3 \neq T_3) \vee (P_2 \neq T_4)$ $C_3 = maxNC \leq RC,$ $maxNC = P_1.size + P_2.size + P_3.size = 9$ $C_{4_2} = maxNP_2 \leq M_2,\ maxNP_2 = 3$ $C_{4_3} = maxNP_3 \leq M_3,\ maxNP_3 = 2$ $C_{4_4} = maxNP_4 \leq M_4,\ maxNP_4 = 1$ $C_{6_1} = 4 - 3 - (P_2 == T_2) \geq 0$ $C_{9_{1\ 4}} = (P_1 \neq T_4),\ C_{9_{4\ 4}} = (P_4 \neq T_4)$
6	P_2 P_3 P_4	$D_2 = \{T_2, T_4\}$ $D_3 = \{T_2, T_3, T_4\}$ $D_4 = \{T_1, T_2, T_3, T_4\}$	$C_{1_1} = alldiff(P_2, P_3),\ C_{1_2} = alldiff(P_3, P_4)$ $C_{1_{2\ 3\ 2\ 3}} = (P_2 \neq T_2) \vee (P_3 \neq T_3)$ $C_{2_{3\ 4\ 3\ 2}} = (P_3 \neq T_3) \vee (P_2 \neq T_4)$ $C_3 = maxNC \leq RC,$ $maxNC = P_2.size + P_3.size = 7$ $C_{4_2} = maxNP_2 \leq M_2,\ maxNP_2 = 2$ $C_{4_3} = maxNP_3 \leq M_3,\ maxNP_3 = 2$ $C_{4_4} = maxNP_4 \leq M_4,\ maxNP_4 = 1$ $C_{6_1} = 4 - 2 - (P_2 == T_2) \geq 0,\ C_{9_{4\ 4}} = (P_4 \neq T_4)$
7	P_2 P_3 P_4 P_5	$D_2 = \{T_2, T_4\}$ $D_3 = \{T_2, T_3, T_4\}$ $D_4 = \{T_1, T_2, T_3, T_4\}$ $D_5 = \{T_2, T_4\}$	$C_{1_1} = alldiff(P_2, P_3, P_5),\ C_{1_2} = alldiff(P_3, P_4)$ $C_{1_{2\ 3\ 2\ 3}} = (P_2 \neq T_2) \vee (P_3 \neq T_3)$ $C_{1_{5\ 3\ 2\ 3}} = (P_5 \neq T_2) \vee (P_3 \neq T_3)$ $C_{2_{3\ 4\ 3\ 2}} = (P_3 \neq T_3) \vee (P_2 \neq T_4)$ $C_{2_{3\ 4\ 3\ 2}} = (P_3 \neq T_3) \vee (P_5 \neq T_4)$ $C_3 = maxNC \leq RC,$ $maxNC = P_2.size + P_3.size + P_5.size = 11$ $C_{4_2} = maxNP_2 \leq M_2,\ maxNP_2 = 3$ $C_{4_3} = maxNP_3 \leq M_3,\ maxNP_3 = 3$ $C_{4_4} = maxNP_4 \leq M_4,\ maxNP_4 = 2$ $C_{6_1} = 4 - 3 - (P_2 == T_2) - (P_5 == T_2) \geq 0$ $C_{9_{4\ 4}} = (P_4 \neq T_4)$

variables and domains, and augment the number and the scope of constraints; no-shows (c_5) remove variables and domains, and reduce the number and the scope of constraints; late arrivals (c_6), late finishes (c_7), and changes in size (c_8) require changes in the constraint description. This example does not comprise changes of type c_9, as they would have the same representation and effect than walk-ins. Note that the allocations of parties P_2 and P_6 become fixed at time $t = 0$, to represent that the customers have been seated and have started their meal. Similarly, party P_3 arrives and gets seated (and fixed) at time $t = 2$.

Apart from the second walk-in (P_7) that was rejected, the other changes were all found feasible, i.e. for each change we found a new seating plan accommodating the change without delaying any meal. However, note that other changes could have been infeasible. For example, if party P_3 (size 3, start 1) arrived on time rather than at time 2, and in a group of 5 rather than 2 people, no solution would have been possible without introducing any delay.

Unavoidable delays can be triggered by changes such as a late arrival (c_6), a late finish (c_7), an increase in size (c_8), or an unexpected booking (c_9). These four types of changes can be critical, i.e. they can often disrupt other diners, and create a financial loss for the restaurant, but the manager must accommodate them even though they depend solely on customer behavior. For our example, the floor manager could ask party P_3 (size 5, start 1) to wait until time 2, when party P_2 is expected to free both tables T_2 and T_3 (which joined can serve the 5 people). Further, when P_6 becomes a late finish, party P_4 (size 2) would then be required to go into table T_4 (size 4), violating the constraint on oversized tables $C_{9_{4,4}}$. Note that, in the example of Figure 6.2, P_4 is finally a no-show, so the actual allocation would not violate the constraint.

Our current model notifies the user whether or not a change can be accommodated without delays, and provides a possible seating plan if one exists. However, the model does not consider an automatic introduction of delays to manage critical changes when they are infeasible (which can only happen in the floor phase, and for changes of type c_6, c_7, c_8, and c_9).

Party	Size	Start	End	Table
P_2	4	0	2	?
P_3	3	1	3	?
P_4	2	2	4	?
P_2	4	0	2	?
P_3	3	1	3	?
P_4	2	2	4	?
P_6	2	0	2	?
P_2	4	0	2	$T_{2,\,3}$
$P_{3(4)}$	3(2)	1(2)	3(4)	?(?)
P_6	2	0	2	T_1
P_7	2	1	3	?
P_2	4	0	2	$T_{2,\,3}$
P_3	3	2	4	?
P_4	2	2	4	?
P_6	2	0	2	T_1
P_2	4	0	2	$T_{2,\,3}$
P_3	3	2	4	?
P_4	2	2	4	?
P_6	2	0	3	T_1
P_2	4	0	2	$T_{2,\,3}$
P_3	3	2	4	?
P_6	2	0	3	T_1
P_2	4	0	2	$T_{2,\,3}$
P_3	2	2	4	?
P_6	2	0	3	T_1

Table[sz]	0	1	2	3
$T_1[2]$			P_4	P_4
$T_2[3]$	P_2	P_2		
$T_3[3]$	P_2	P_2		
$T_4[4]$		P_3	P_3	
$T_1[2]$	P_6	P_6	P_4	P_4
$T_2[3]$	P_2	P_2		
$T_3[3]$	P_2	P_2		
$T_4[4]$		P_3	P_3	
$T_1[2]$	P_6	P_6	P_4	P_4
$T_2[3]$	P_2	P_2		
$T_3[3]$	P_2	P_2		
$T_4[4]$		P_3	P_3	
$T_1[2]$	P_6	P_6	P_4	P_4
$T_2[3]$	P_2	P_2		
$T_3[3]$	P_2	P_2		
$T_4[4]$			P_3	P_3
$T_1[2]$	P_6	P_6	P_6	
$T_2[3]$	P_2	P_2	P_4	P_4
$T_3[3]$	P_2	P_2		
$T_4[4]$			P_3	P_3
$T_1[2]$	P_6	P_6	P_6	
$T_2[3]$	P_2	P_2		
$T_3[3]$	P_2	P_2		
$T_4[4]$			P_3	P_3
$T_1[2]$	P_6	P_6	P_6	
$T_2[3]$	P_2	P_2	P_3	P_3
$T_3[3]$	P_2	P_2		
$T_4[4]$				

Figure 6.2: Example of dynamic problem on the floor phase: sequence of static instances (left); possible seating plans (right).

Table 6.2: CSP representation for the example of Figure 6.2. Note: (RC, M_2, M_3, M_4) = (12, 4, 3, 2).

i	X'^i	D'^i	C'^i
1	P_2 P_3 P_4 P_5 P_6	$D_2 = \{T_2, T_4\}$ $D_3 = \{T_2, T_3, T_4\}$ $D_4 = \{T_1, T_2, T_3, T_4\}$ $D_5 = \{T_2, T_4\}$ $D_6 = \{T_1, T_2, T_3, T_4\}$	$C_{1_1} = alldiff(P_2, P_3, P_6),\ C_{1_2} = alldiff(P_3, P_4)$ $C_{1_{2\ 3\ 2\ 3}} = (P_2 \neq T_2)\ \vee\ (P_3 \neq T_3)$ $C_{1_{2\ 6\ 2\ 3}} = (P_2 \neq T_2)\ \vee\ (P_6 \neq T_3)$ $C_{2_{3\ 4\ 3\ 2}} = (P_3 \neq T_3)\ \vee\ (P_2 \neq T_4)$ $C_3 = maxNC \leq RC,$ $maxNC = P_2.size + P_3.size + P_6.size = 9$ $C_{4_i} = maxNP_i \leq M_i,$ $maxNP_2 = 3,\ maxNP_3 = 2,\ maxNP_4 = 1$ $C_{6_1} = 4 - 3 - (P_2 == T_2) \geq 0$ $C_{9_{4\ 4}} = (P_4 \neq T_4),\ C_{9_{6\ 4}} = (P_6 \neq T_4)$
2	P_2 = T_2 P_3 P_4 P_5 P_6 = T_1 P_7	$D_2 = \{T_2, T_4\}$ $D_3 = \{T_2, T_3, T_4\}$ $D_4 = \{T_1, T_2, T_3, T_4\}$ $D_5 = \{T_2, T_4\}$ $D_6 = \{T_1, T_2, T_3, T_4\}$ $D_7 = \{T_1, T_2, T_3, T_4\}$	$C_{1_1} = alldiff(P_2, P_3, P_6, P_7),$ $C_{1_2} = alldiff(P_3, P_4, P_7)$ $C_{1_{2\ 3\ 2\ 3}} = (P_2 \neq T_2)\ \vee\ (P_3 \neq T_3)$ $C_{1_{2\ 6\ 2\ 3}} = (P_2 \neq T_2)\ \vee\ (P_6 \neq T_3)$ $C_{1_{2\ 7\ 2\ 3}} = (P_2 \neq T_2)\ \vee\ (P_7 \neq T_3)$ $C_{2_{3\ 4\ 3\ 2}} = (P_3 \neq T_3)\ \vee\ (P_2 \neq T_4)$ $C_3 = maxNC \leq RC,$ $maxNC = P_2.size + P_3.size + P_6.size + P_7.size = 11$ $C_{4_i} = maxNP_i \leq M_i,$ $maxNP_2 = 4,\ maxNP_3 = 2,\ maxNP_4 = 1$ $C_{6_1} = 4 - 4 - (P_2 == T_2) \geq 0$ $C_{9_{4\ 4}} = (P_4 \neq T_4),\ C_{9_{6\ 4}} = (P_6 \neq T_4)$ $C_{9_{7\ 4}} = (P_7 \neq T_4)$
3	P_2 = T_2 P_3 (start=2, end=4) P_4 P_5 P_6 = T_1	$D_2 = \{T_2, T_4\}$ $D_3 = \{T_2, T_3, T_4\}$ $D_4 = \{T_1, T_2, T_3, T_4\}$ $D_5 = \{T_2, T_4\}$ $D_6 = \{T_1, T_2, T_3, T_4\}$	$C_{1_1} = alldiff(P_2, P_6),\ C_{1_2} = alldiff(P_3, P_4)$ $C_{1_{2\ 6\ 2\ 3}} = (P_2 \neq T_2)\ \vee\ (P_6 \neq T_3)$ $C_3 = maxNC \leq RC,$ $maxNC = P_2.size + P_6.size = 6$ $C_{4_i} = maxNP_i \leq M_i,$ $maxNP_2 = 2,\ maxNP_3 = 1,\ maxNP_4 = 1$ $C_{6_1} = 4 - 2 - (P_2 == T_2) \geq 0$ $C_{9_{4\ 4}} = (P_4 \neq T_4),\ C_{9_{6\ 4}} = (P_6 \neq T_4)$
4	P_2 = T_2 P_3 P_4 P_5 P_6 = T_1 (end=3)	$D_2 = \{T_2, T_4\}$ $D_3 = \{T_2, T_3, T_4\}$ $D_4 = \{T_1, T_2, T_3, T_4\}$ $D_5 = \{T_2, T_4\}$ $D_6 = \{T_1, T_2, T_3, T_4\}$	$C_{1_1} = alldiff(P_2, P_6),\ C_{1_2} = alldiff(P_3, P_4, P_6)$ $C_{1_{2\ 6\ 2\ 3}} = (P_2 \neq T_2)\ \vee\ (P_6 \neq T_3)$ $C_3 = maxNC \leq RC,$ $maxNC = P_3.size + P_4.size + P_6.size = 7$ $C_{4_i} = maxNP_i \leq M_i,$ $maxNP_2 = 3,\ maxNP_3 = 1,\ maxNP_4 = 1$ $C_{6_1} = 4 - 2 - (P_2 == T_2) \geq 0$ $C_{9_{4\ 4}} = (P_4 \neq T_4),\ C_{9_{6\ 4}} = (P_6 \neq T_4)$
5	P_2 = T_2 P_3 P_5 P_6 = T_1	$D_2 = \{T_2, T_4\}$ $D_3 = \{T_2, T_3, T_4\}$ $D_5 = \{T_2, T_4\}$ $D_6 = \{T_1, T_2, T_3, T_4\}$	$C_{1_1} = alldiff(P_2, P_6),\ C_{1_2} = alldiff(P_3, P_6)$ $C_{1_{2\ 6\ 2\ 3}} = (P_2 \neq T_2)\ \vee\ (P_6 \neq T_3)$ $C_3 = maxNC \leq RC,$ $maxNC = P_2.size + P_6.size = 6$ $C_{4_i} = maxNP_i \leq M_i,$ $maxNP_2 = 2,\ maxNP_3 = 1,\ maxNP_4 = 1$ $C_{6_1} = 4 - 2 - (P_2 == T_2) \geq 0$ $C_{9_{6\ 4}} = (P_6 \neq T_4)$
6	P_2 = T_2 P_3 (size=2) P_5 P_6 = T_1	$D_2 = \{T_2, T_4\}$ $D_3 = \{T_1, T_2, T_3, T_4\}$ $D_5 = \{T_2, T_4\}$ $D_6 = \{T_1, T_2, T_3, T_4\}$	$C_{1_1} = alldiff(P_2, P_6),\ C_{1_2} = alldiff(P_3, P_6)$ $C_{1_{2\ 6\ 2\ 3}} = (P_2 \neq T_2)\ \vee\ (P_6 \neq T_3)$ $C_3 = maxNC \leq RC,$ $maxNC = P_2.size + P_6.size = 6$ $C_{4_i} = maxNP_i \leq M_i,$ $maxNP_2 = 3,\ maxNP_3 = 1,\ maxNP_4 = 1$ $C_{6_1} = 4 - 2 - (P_2 == T_2) \geq 0$ $C_{9_{3\ 4}} = (P_3 \neq T_4),\ C_{9_{6\ 4}} = (P_6 \neq T_4)$

6.3 Solution process

Figure 6.3 represents the solution process over a general sequence of changes, for either booking or floor management. SP_0 is the initial seating plan. SP_N ($N = 1, 2, 3, ..$) corresponds to the seating plan updated after solving the problem instance generated by change CH_N, which occurs at time t_N. Note that, when change CH_N occurs, all allocations of parties which have been seated at any time $t < t_N$ cannot be changed. In our model, if CH_N is found feasible then SP_N contains (or accommodates) the change; otherwise, CH_N is rejected, and the preceding seating plan is restored, i.e. $SP_N = SP_{N-1}$.

For simplicity, we represent a dinner session with a time window 0 .. T, i.e. 0 and T are the first and last available time to serve a meal. Restaurant table management starts with a booking phase, anticipating the dinner session, followed by a floor management phase, covering the entire dinner session. Then, any change CH_N can occur at any time $t_N < 0$ in the booking phase, and at any time $0 \leq t_N \leq T$ in the floor management phase.

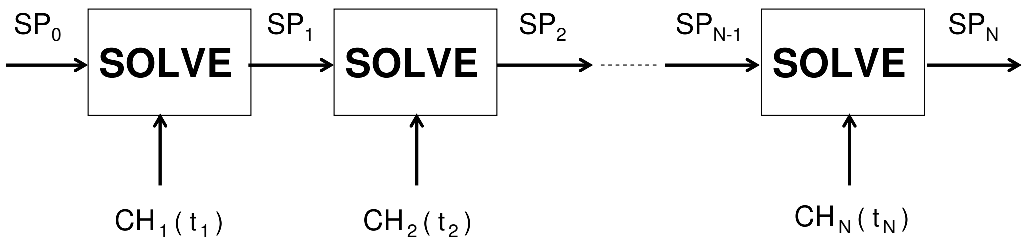


Figure 6.3: Diagram representing the solution process of the dynamic problem: to each change CH_N corresponds a new problem instance.

6.3.1 Solvers

During the solution process, seating plans are updated according to the decisions taken by a solver ("SOLVE" blocks in Figure 6.3). We implemented two versions of solver based on the modelling we have done in the previous chapters (we name them $SOLVER_1$ and $SOLVER_2$), and two other versions imitating the traditional allocation methods adopted in restaurants (which we name $SOLVER_3$ and

$SOLVER_4$). Specifically, to search whether (and how) a change CH_N can be accommodated into a given seating plan SP_N:

- $SOLVER_{1,\,2}$ can re-allocate anyone in SP_N who has not sat down;

- $SOLVER_{3,\,4}$ cannot re-allocate anybody in SP_N.

For $SOLVER_{1,2}$, all parties which are planned to be seated at any time t in the future (between t_N and T) have not been committed to their original table. In particular, during the booking phase all parties are free for reallocation, as $t_N < 0$. For $SOLVER_{3,4}$, all parties are instead committed to their original table.[†]

All solvers are based on our constraint satisfaction model of the problem (Chapter 4 and 5). $SOLVER_1$ and $SOLVER_3$ stop after the first solution is found (or the problem is found infeasible). $SOLVER_2$ and $SOLVER_4$ extend the first two versions by performing optimization - i.e. after the first solution is found, they try to search alternative and more flexible seating plans, based on our optimization model (Chapter 5).

Later in the chapter all these versions will be tested and compared over simulated booking and floor management sessions. We expect that the ability of $SOLVER_{1,2}$ to perform reallocations is going to allow more flexibility for the accommodation of new requests, and also more robustness in absorbing changes (e.g. delays) without introducing other delays. Further, starting from (and maintaining) a more flexible seating plan ($SOLVER_{2,4}$) is also expected to be important in order to maximize the reservation potential - by increasing the chances to accept future bookings or walk-ins. Flexibility can also be seen as a form of robustness, i.e. a more flexible seating plan should improve the ability of the restaurant to absorb changes without causing delays or disruptions.

[†]In fact, current booking systems are based on paper, i.e. reservations are taken by writing party names into table slots in a booking sheet. After a reservation has been written into a table the allocation is rarely changed - it is inconvenient to do the operation on paper, and simultaneously reallocating even a few reservations can be too complex, especially on a crowded booking sheet.

6.4 Seating plan stability constraining the number of changes

Before moving to the experiments, we now describe a final tuning on the search algorithms utilized for solving the dynamic scheduling problem. The constraint satisfaction and optimization models described in the previous chapters do not consider the number of table reallocations from one seating plan to the next - their aim is to find any (improving) plan. During the floor management phase, however, too many changes cause confusion in the restaurant, making it difficult for staff to understand and evaluate each solution. Frequent changes in table configurations can cause disturbance also to the customers who are eating - if the tables near where they got seated have been moved several times. Therefore, table management should, when possible, try to maintain the stability of the plan, and should prefer new plans with few changes.

We extended the previous models, so that when changes occur, we search for new solutions in two phases: first, we search in the neighbourhood of the preceding 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 in Figure 6.4.

```
   solution   =   original
discrepancy   =   0

      while       ( (timer < $timeout_1$) && (discrepancy < $discrepancy_{MAX}$) )
              if  Solver.solve(CSP, MH, $timeout_1$, original, discrepancy) == true
                  solution = Solver.getSolution()
                  return solution
            else  discrepancy += 1

         if       Solver.solve(CSP, MH, $timeout_2$, original, any) == true
                  solution = Solver.getSolution()

     return       solution
```

Figure 6.4: Basic algorithm for stable solutions.

The maximum 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 if the new change 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 maximum number of changes. The *solve* procedure is extended to include the discrepancy limit, which is posted as a constraint $C_{DISCREPANCY}$ on the decision variables, i.e. the set of parties $P_1, P_2, .., P_N$:

$$C_{DISCREPANCY}:$$
$$discrepancy.old < \sum_{n=1..N} (P_n \neq original.getValue(P_n)) \leq discrepancy$$

The constraint restricts the number of changes to be greater than the discrepancy value at the previous loop, i.e. *discrepancy.old*.[†] A similar procedure is applied when searching for flexible solutions - the parameters *discrepancyMax*, *timeout*$_1$, and *timeout*$_2$, allow stability to be traded for flexibility.

6.5 Objective

We want to evaluate our solutions (i.e. $SOLVER_{1,2}$) over simulated booking and floor management sessions, comparing to our imitations of the traditional allocation method adopted in the restaurants (i.e. $SOLVER_{3,4}$). Our higher level goal is to verify that our techniques are ***efficient*** for a practical use in a real restaurant, and provide more ***flexible*** and ***robust*** solutions than traditional approaches - so they can effectively represent an improvement in the ability to solve the dynamic problem. In relation to this, we are going to present a number of experiments focusing on the following questions:

- Does the fact that parties are free for reallocation until the time they get seated allow more flexibility to manage future table demand and delays? In particular, does this freedom allow us to seat more covers by the end of the

[†]In this case *discrepancy* is incremented by 1 at each loop, so *discrepacy.old* = *discrepancy*-1; however, the discrepancy count could also be increased in higher steps.

night, reducing the number of meal requests which get turned down, or to minimize the cases where the propagation of a current delay causes more delays over future meals?

- Does our solution provide more flexible seating plans? In particular, by using flexibility do we get more covers by the end of the night?
- How does our solution make use of diners' start time flexibility, i.e. can the optimization model be used to sell more flexible times when the customer has no specific preference? In particular, if we use the flexibility model over start times, do we get more covers at the end of the night?
- Does the reasoning on future knowledge included into our model improve seating plan flexibility? Again, does it allow us to seat more people by the end of the night?

The booking and floor sessions we simulate are analogous to the examples previously discussed, but are dimensioned over a real size restaurant. For simplicity, we will represent booking sessions with sequences of booking requests, and floor sessions with sequences of walk-in requests, late finishes, or late arrivals.

6.6 Booking simulation

We represent an instance of a single booking session as a set of 100 booking requests (or parties) arriving one by one over time. Therefore, to simulate a booking session we generate an ordered set of booking requests - each request is assigned a distinct identification number id, ranging from 1 to 100, to reflect the arrival order. We consider requests for parties of size in the range 1 to 8, using a realistic distribution retrieved from real data collected in the restaurant Eco (i.e. from past booking sheets). Specifically, considering the 8 possible sizes in increasing order, the frequency we assign to each is respectively 1%, 43%, 15%, 14%, 12%, 9%, 6%, 5%. Start times are generated in the range 0 to 24 (i.e. 4 p.m. to 10 p.m. discretized in 15-minute units). In the restaurant Eco the standard dinner duration is estimated to 2 hours, which means that the general policy applied by the booker is to reserve a two hour slot to each booking. For particularly large parties dining at peak times he may assign longer durations, and for small parties dining off peak he may give a shorter slot. Considering a first approximation of the standard booking process, for each generated party we generate (and assume) a duration in the range 7 to 9 (i.e. 1.45 hours to 2.15 hours, discretized in 15-minutes units), with average 8 (i.e. 2:00 hours).

For our experiments, we generate and tests groups of 30 booking sessions each, to represent a month of booking management. We test $SOLVER_{1,\ 2}$ (our solutions) and $SOLVER_{3,\ 4}$ (our simulations of traditional solutions) using the diagram discussed in Section 6.2, Figure 6.3. For instance, we consider each set of 100 booking requests as a list of changes CH_1, CH_2, .., CH_{100}. We start from an empty seating plan $SP_0 = \emptyset$, and process the changes in the order. For each request, we let the solver run for a time of 10 seconds - which is a reasonable time to evaluate the *efficiency* for a practical use in a real restaurant. If no solution is found within 10 seconds the current request gets rejected and the preceding seating plan is restored. $SOLVER_1$ and $SOLVER_3$ only search for a first feasible solution. $SOLVER_2$ and $SOLVER_4$ also perform optimization, so, if they find a first feasible solution before the 10 seconds, they use the remaining time to search for better solutions.

6.6.1 Results on standard booking

Table 6.3 and Figure 6.5 show the results of the first experiments, concerning a standard booking simulation (as described above). The table reports the average number of covers reached after 60 requests, the number of times (over the 30 booking sessions) the reservation target was achieved within 60 requests, and the average number of requests to achieve the reservation target. The graph represents the mean number of accepted covers over number of requests. Each point is an average over the 30 booking sessions. The two lower curves represents the traditional way to take bookings, while the two on the top are our solutions. The horizontal line at 180 covers is the target, i.e. the restaurant is happy about an evening session if the final turnover of people is above 180.

As we can see, both our solutions reach the target after an average of $\sim$ 57

Table 6.3: Comparing the four solvers on the reservation target.

	$SOLVER_1$	$SOLVER_2$	$SOLVER_3$	$SOLVER_4$
mean Nb of covers achieved after 60 requests	182.6	182.9	170.9	173.4
Nb of times achieved 180 covers after 60 requests	20/30 (67%)	21/30 (70%)	7/30 (23%)	10/30 (33%)
mean Nb of requests to reach 180 covers	57.5	57.5	67.5	67.5

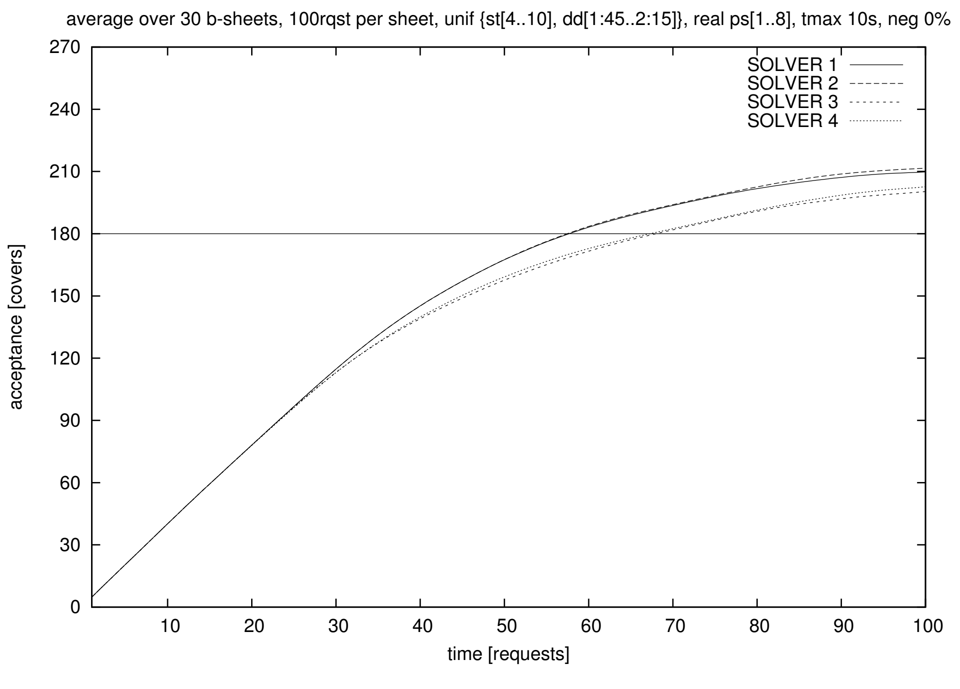


Figure 6.5: Tests on standard booking: mean number of covers over number of requests, comparing $SOLVER_1$, $SOLVER_2$, $SOLVER_3$, and $SOLVER_4$. Reservation target is 180 covers.

requests, while the other solutions require ~ 67. From the restaurant point of view, this means that, on average, if for one night the restaurant receives only 57 requests, then our solver would achieve the target, while the traditional allocation method would still be $\sim 7\%$ below. From the customer point of view, this also means that using our solver we have turned down fewer requests, so the restaurant gets fewer disappointed customers. $SOLVER_1$ and $SOLVER_2$ are also more robust, as they achieve the target in more cases if we fix the maximum number of requests to 60 - i.e. 67% and 70% (in the order), against 23% and 33% of $SOLVER_3$ and $SOLVER_4$.

Comparing the basic version, $SOLVER_1$, to the one extended to perform optimization, $SOLVER_2$, the performance is similar, with the second version slightly better on average. This similarity was expected, as during the booking phase all the reservations already taken can always be reallocated in order to fit a new request, so there is no commitment of tables to parties to be decided and optimized. The fact that $SOLVER_2$ is slightly better suggests that there is a benefit from having a good solution to start with, at each request. In fact, our algorithms starts the search from the neighbourhood of the preceding solution (i.e. the one before the new request arrives), gradually increasing the number of allowed changes. We then expect that, especially as the problem gets larger (and harder), the more flexible the preceding solution is, the more likely we are to find a new solution accommodating the new request (assuming one exists) within the 10 seconds available. Note that the search space is the same for $SOLVER_1$ and $SOLVER_2$, so there would be no difference between using one or the other assuming we had no limit in the search time.

A similar comparison holds between the traditional version $SOLVER_3$ and its extension with optimization $SOLVER_4$, with the second slightly better. In this case, all allocations of tables to parties are fixed at reservation time. It appears there is (again) a slight benefit from performing optimization, i.e. picking a more rather than a less flexible allocation for each request when it arrives.

Figure 6.6 shows the profile of the flexibility measure over time for the four solvers. We observe how in the first part $SOLVER_4$ and especially $SOLVER_2$ preserve a higher level of flexibility, compared to $SOLVER_1$ and $SOLVER_3$. From the results in Figure 6.5, both $SOLVER_1$ and $SOLVER_2$ are able to satisfy more requests compared to the other solvers, therefore for them the restaurant gets crowded earlier - in particular, after 55-60 requests the increase in number of people allocated is always around 10. A more crowded restaurant means that the flexibility to accept future requests is reduced. This explains the fact that in the second part both the curves of flexibility for $SOLVER_1$ and $SOLVER_2$ appear below the curves of the traditional solvers, $SOLVER_3$ and $SOLVER_4$. The effect is particularly evident for $SOLVER_1$, i.e. our version with no flexibility optimization.

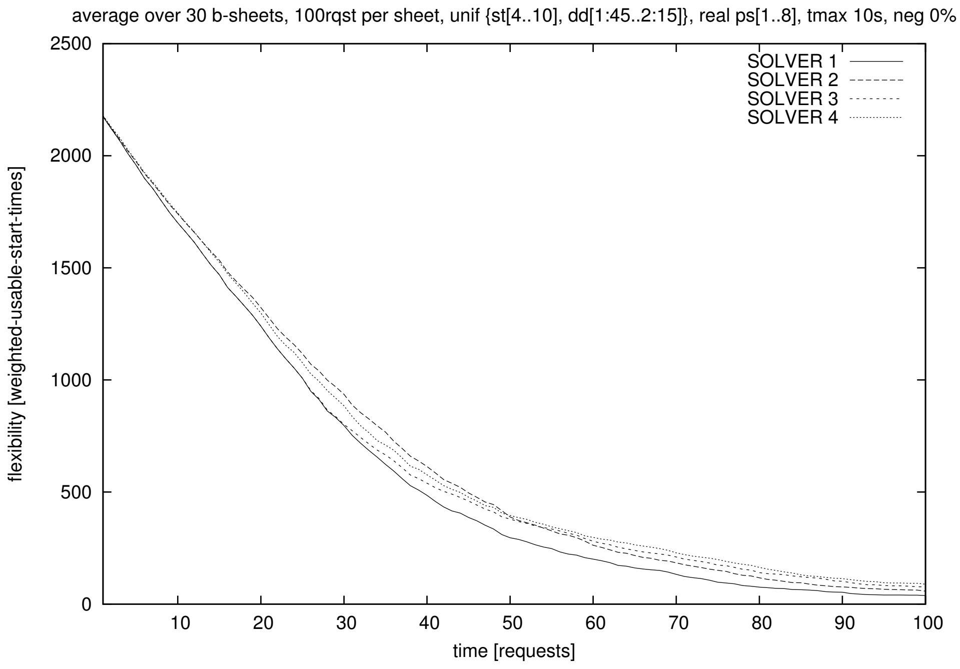


Figure 6.6: Tests on standard booking: flexibility profile over number of requests, for $SOLVER_1$, $SOLVER_2$, $SOLVER_3$, and $SOLVER_4$.

6.6.2 Results on booking with diner's start time flexibility

Sometimes, customers do not have a real preference over a specific booking time, i.e. they are flexible about the time to consume their dinner. In other cases, booking requests for a specific time slot cannot be accommodated because the restaurant is fully booked at that time - so the booker can offer alternative time slots.

In both cases, the booker can look at the current list of bookings and propose the possible times available to the customer.

We now test whether our flexibility-based optimization can be used to help the booker negotiate start times, i.e. suggesting booking times that would maintain more flexible seating plans, for a higher reservation potential, and therefore for a higher final turnover. We repeat the tests over the same 30 booking sessions of 100 requests, using the same distribution over party sizes, start times, and dinner durations as in the previous experiment. This time, however, if *start*$_i$ is the original start time (or booking time) that we generate for each request RQ_i ($i = 1..100$), we assume that those bookings which we represent as customers who are flexible (or available for negotiation) can have a start time equal to any among the three possibilities *start*$_i$, *start*$_i$ + 1 *hour*, and *start*$_i$ - 1 *hour* (for simplicity, we considered only alternative times on the hour). So, to represent for example a customer who would like a table for sometime around 7 o'clock, we will consider 6 o'clock, 7 o'clock, and 8' o'clock as possible start times, and then the solver will optimize and pick the time which preserves a more flexible solution for future requests.

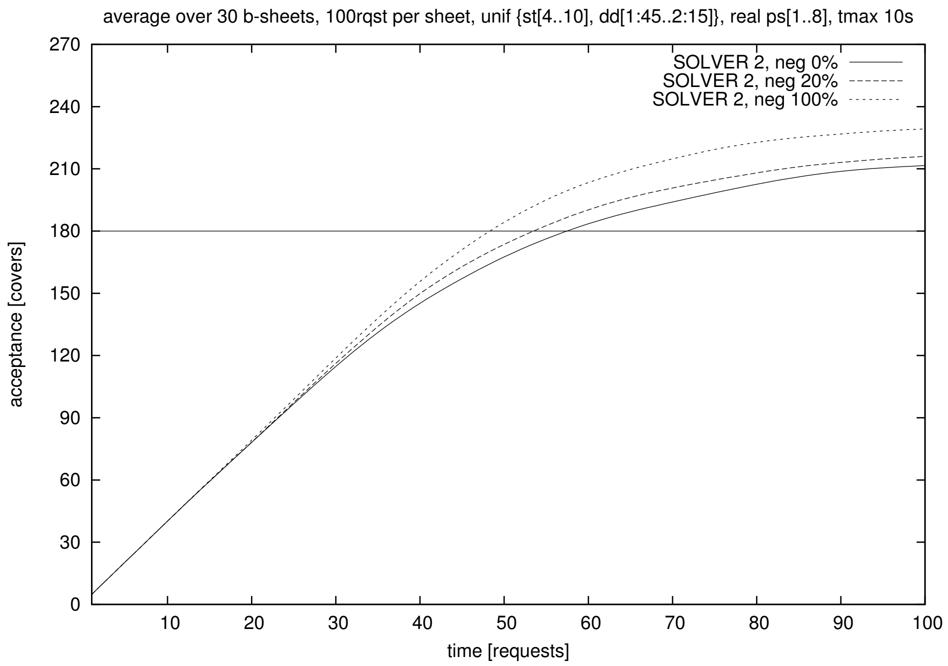
For this experiment we compare three versions of $SOLVER_2$, which is the best solver resulted from the experiment presented in the previous section. The part of optimization performed by $SOLVER_2$ is based on measuring weighted usable start times, i.e. we set the flexibility measure to flexibility$_{WUS}$. For the first of the three versions we assume no start time flexibility (or negotiation). For the second version, we assume one customer out of five (i.e. 20%, selected randomly) is available to accept any of the three possible booking times. Similarly, for the third version we assume all customers (i.e. 100%) would accept any of the three possible start times. This last version may not represent a realistic case, but is used here to give a better understanding of the effect (and the potential benefit) of supporting negotiation. For each new request RQ_i, we run $SOLVER_2$ for 10 seconds (as usual), setting the start time of RQ_i equal to the original request *start*$_i$, and, if the RQ_i is selected for negotiation, we repeat the 10 second run two more times, one with *start*$_i$ + 1 *hour* and one with *start*$_i$ - 1 *hour*.

Figure 6.7 (top) shows the results of this experiment, again, in terms of mean number of accepted covers over number of requests. As we can see, if for each request we allow the solver to choose among alternative times (from the set of three possible, and based on our flexibility measure), the number of requests necessary to get to the reservation target of 180 covers is significantly reduced. For instance, assuming 100% of customers are willing to change their start time by one hour, the resulting mean number of requests to achieve the target is reduced to ~ 47, i.e. 10 fewer than the case with 0% negotiation (and 20 fewer than the performance of the traditional solver $SOLVER_4$). Even though in reality not all customers are so flexible on the start time, this result shows how the restaurant could make a large improvement by adopting negotiation as a systematic strategy. Assuming for example the total number of requests for the night is 57, then we can see that with 0% negotiation the restaurant has bookings for 180 covers, while allowing 100% negotiation it has bookings for more than 200 covers, i.e. the increase is greater than 10%. Encouraging the customer to choose dinner slots which better preserve flexibility for future requests can significantly increase the number of people which can be accommodated.

The intermediate case, with 20% of bookings available for negotiation, represents a good approximation of what really happens in the restaurants.[†] Comparing this case against the case with 0% negotiation we can still observe a significant impact of allowing some spontaneous (rather than systematic) negotiation. For instance, with a 20% negotiation frequency the reservation target is achieved after an average of 53 requests, 4 (i.e. $\sim 7\%$) fewer than if we take bookings without negotiation.

Figure 6.7 (bottom) shows the flexibility profile over time (or over number of requests) for the three levels of negotiation. By allowing negotiation the solver is able to decide which is the best start time for each request in order to preserve a higher flexibility for future requests. In the first part of the graph the increase in flexibility gained by allowing negotiation is clear, and proportional to the level of negotiation. In the second part the three curves of flexibility are more levelled. This is due to the fact that a versions with a higher level of negotiation has been allocating more parties - so the restaurant occupancy is more crowded and as a consequence the flexibility drops.

[†]In fact, in the mornings and afternoons I spent at the reservation office of the restaurant Eco, I estimated that about one phone call out of 5 regards either a customer asking for a table for a time already fully booked, or a customer asking which times are still available to get a table.



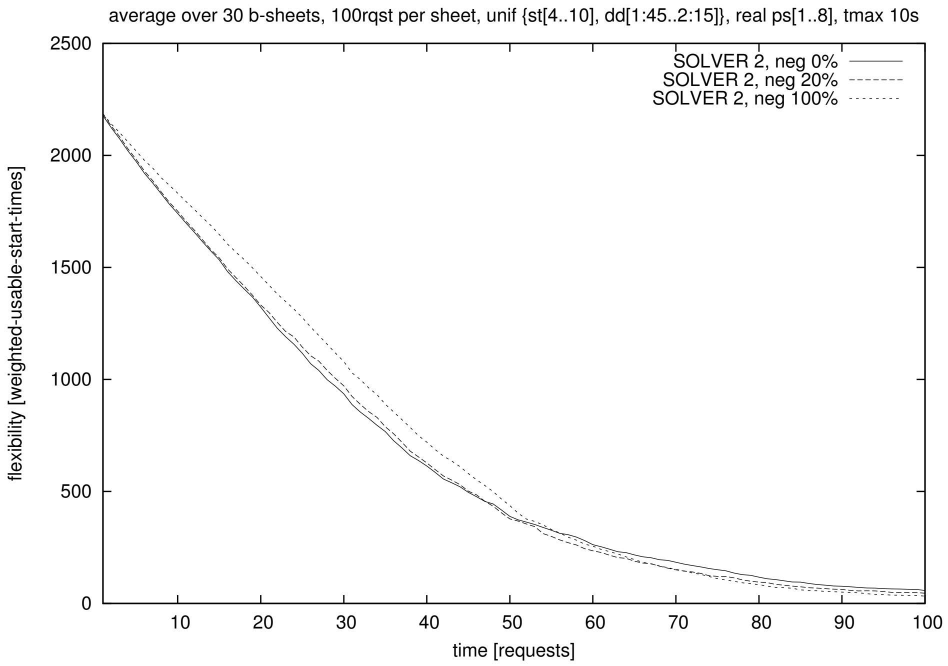


Figure 6.7: Tests on booking with negotiation. Mean number of covers over number of requests (top). Flexibility profile over number of requests (bottom). Comparing $SOLVER_2$ for different levels of negotiation.

6.6.3 Results on booking for different flexibility measures

In Section 5.5, we implemented three flexibility measures: flexibility$_{WUS}$, based on usable start times; flexibility$_{WDZ}$, based on dead zones; and flexibility$_{WNS}$, based on the potential number of seatings. All measures weight each table slot by the table capacity. So far, in our tests we have only considered optimization based on flexibility$_{WUS}$. We now want to compare the three flexibility measures. Thus, we test three versions of $SOLVER_2$: the first maximizes usable start times; the second minimizes dead-zones; and the third maximizes the potential number of seatings. Results are reported in Figure 6.8. The problems we tested are the same as in the previous experiments, i.e. 30 booking sheets of 100 requests each, with same distribution over party sizes, start times, and durations. Comparing the number of accepted covers achieved by flexibility$_{WUS}$ to the number concerning the other two measures we can see a very little difference: flexibility$_{WUS}$ appears overall slightly better, with a peak of improvement of ~ 1 cover in the range 60 to 70 requests, followed by flexibility$_{WNS}$, and then by flexibility$_{WDZ}$. However, considering a typical turnover of 180-200 covers, the improvement is not significant. We can conclude that the three measures have a similar performance, particularly in the range 0 to 40 requests, and after 80 requests.

6.6.4 Results on booking using future knowledge

In the previous experiments, we tested booking sessions with start times uniformly distributed in the range 0 to 24, i.e. from 4 p.m. to 10 p.m. For our flexibility measure (i.e. flexibility$_{WUS}$), this means that all possible start times were equally likely and therefore were given the same weight in the count of flexibility. In order to evaluate how our flexibility estimates model future knowledge and how our optimization algorithm makes use of it, we now consider booking sessions with start times distributed according to specific patterns. Specifically, we generate 30 booking sessions of 100 requests each, with start times taken from a distribution based on a booking pattern for a Sunday session at the Eco restaurant. We then feed the distribution into our measure - so the measure gives different weights to different booking times according to that pattern (as described in Section 5.5).

Figure 6.9 (top) compares the performance of $SOLVER_2$ considering four

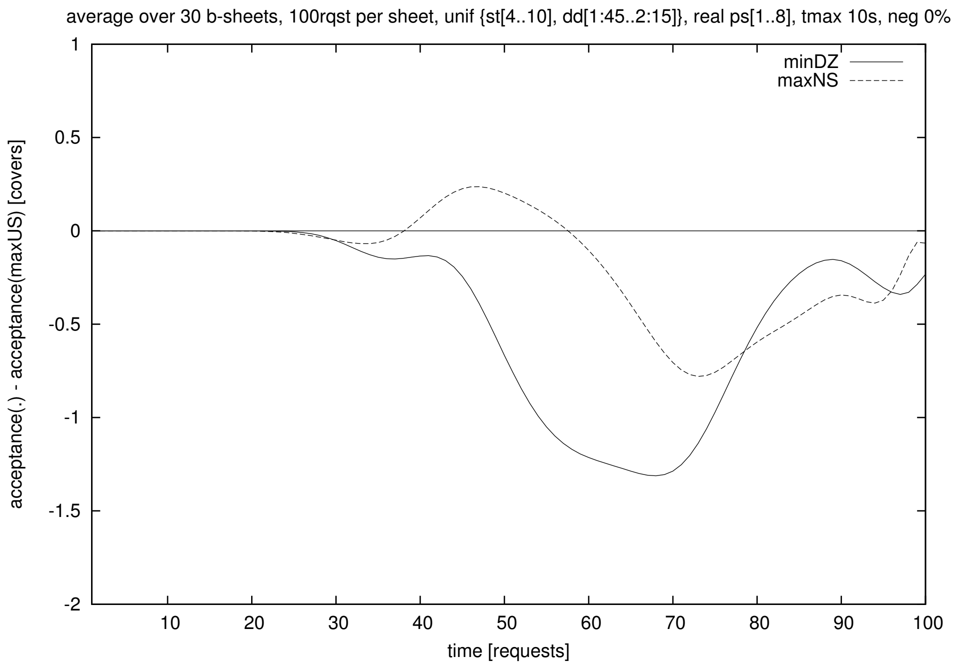


Figure 6.8: Tests on standard booking: mean number of covers over number of requests, comparing $SOLVER_2$ over the different measures of flexibility.

different versions: (i) with no negotiation, and assuming an incorrect (i.e. uniform) booking pattern; (ii) with no negotiation, and with the correct (i.e. Sunday) booking pattern; (iii) with 20% negotiation, and with the incorrect (i.e. uniform) booking pattern; and (iv) with 20% negotiation, and with the correct (i.e. Sunday) booking pattern. For comparison we also report the performance of $SOLVER_4$ (with no negotiation and no booking pattern). We can observe how version (i) and (ii) are almost overlapping with each other, i.e. there is little benefit from using the correct booking pattern rather than a uniform weighting into the flexibility computation. However, comparing the corresponding versions (iii) and (iv), both allowing some booking negotiation, we can observe how the version using the correct pattern is now noticeably better than the one using uniform weights - Figure 6.9 (bottom) shows a zoom over the second half of the curves, where we estimate an improvement of $\sim 1\%$. This represents a small improvement, but it suggests that, if we allow negotiation then the use of a correct booking pattern can effectively make some difference.

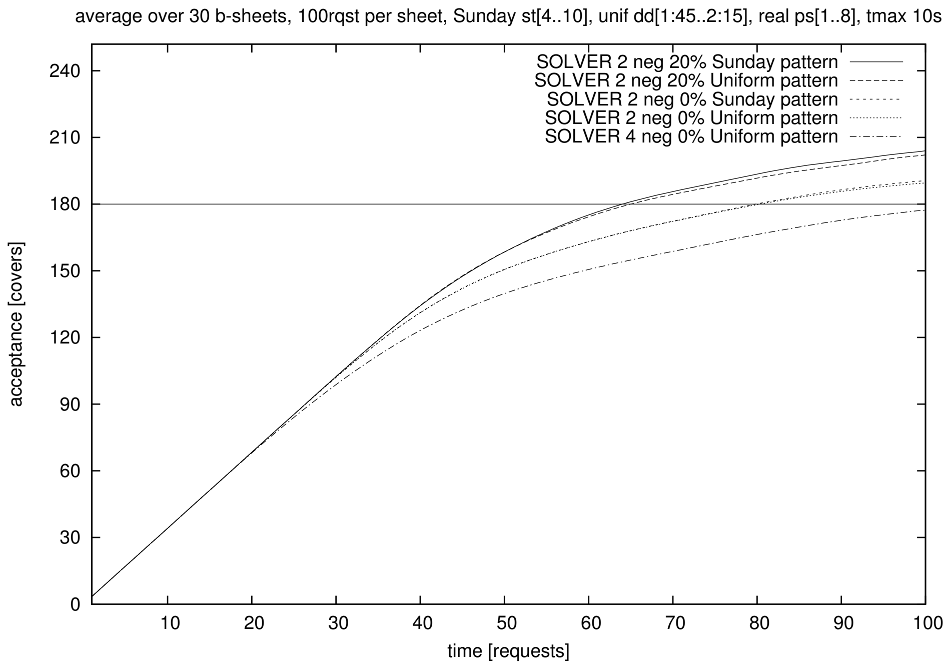
Finally, note that each version achieves the target of 180 covers later compared to the previous experiments on uniform bookings. In fact, as bookings are now not distributed uniformly (i.e. according to our Sunday pattern, bookings are more concentrated between 4 p.m. and 7 p.m.), because of the contention between 4 p.m. and 7 p.m. more reservations must be turned down, and so it takes more reservations to meet the target.

This is our first attempt at including future knowledge, and the issue needs to be further investigated (e.g. computing better flexibility estimates). For example, table slots should be weighted not simply by start time distribution but instead by the combined distribution of party size and start time. In that case, the new booking pattern would be a two dimensional surface (i.e. f(size, time), which can again be retrieved from past booking sheets).

6.7 Floor management simulation on walk-ins

To represent an instance of a single floor management session we use an ordered set of 70 booking requests (or parties), similarly to the simulation performed for the booking phase. In this case, we assume we are at the beginning of the dinner session, the first 40 parties in the set are customers who have reserved on the day before, while the last 30 are parties who are going to walk in during the night and request a table (40 and 30 are realistic numbers). The last 30 parties are ordered by increasing start time, to simulate the real situation - with walk-ins arriving in chronological order. Considering the diagram presented in Figure 6.3, the initial seating plan SP_0 now contains the first 40 parties, and the next seating plans will then come from solving the sequence of 30 instances generated by the sequence of 30 walk-ins. As usual, we represent the sequence of walk-in requests as changes CH_1, CH_2, ..., CH_{30}. The main difference from the booking phase is that now, when a walk-in CH_i arrives at time t_i during the dinner session, none of the parties who have started their meal can be reallocated.

Figure 6.10 (top) shows the results achieved by the four solvers, again in terms of accepted covers over number of requests (now walk-ins). The 40 parties that are already in the initial seating plan before the first walk-in turns up contribute to an initial load of about 150 covers. Therefore, the new target for our dinner



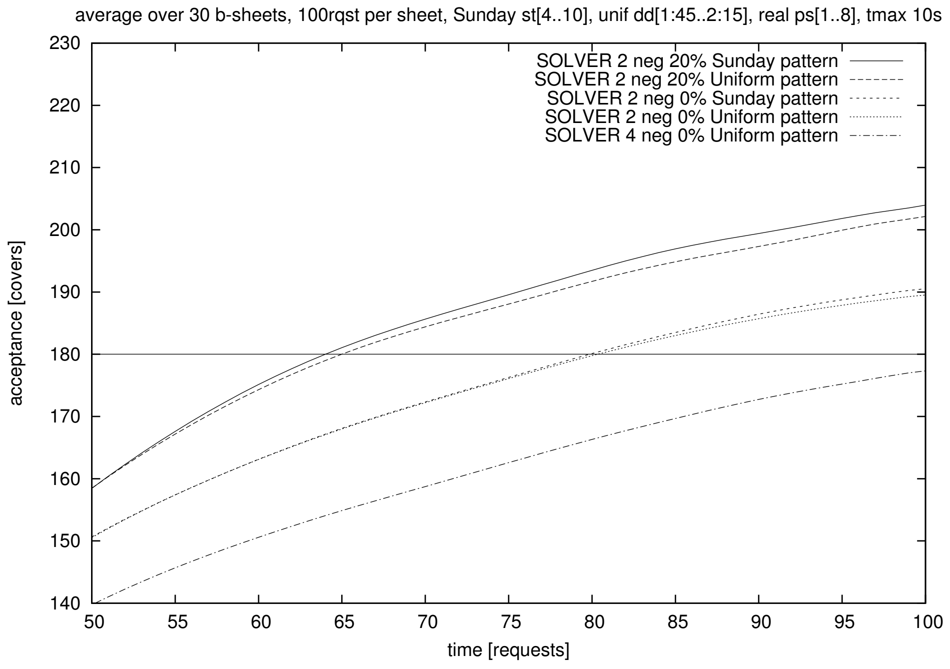
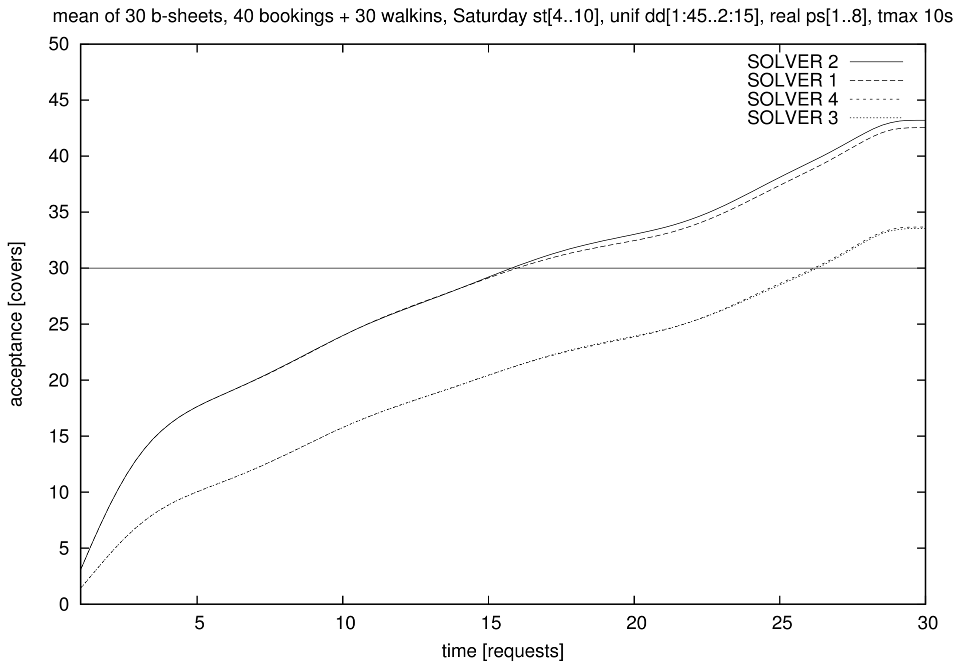


Figure 6.9: Tests on standard booking: mean number of parties over number of requests, comparing $SOLVER_2$ with and without using future knowledge.

session is to achieve 30 extra covers (represented by the horizontal line), which would get us to the original target of 180 covers overall. From the graphs we can see how $SOLVER_1$ and $SOLVER_2$ significantly outperform the traditional methods $SOLVER_3$ and $SOLVER_4$, similar to what we found in the tests for the booking phase. Note that the difference in performance increases rapidly at the beginning, where there is more freedom for reallocations, while later, when fewer parties can be reallocated, the gap is almost constant. The first two solvers reach the target after ~ 16 walk-ins, while the second two require ~ 26, i.e. 10 more. If for example the restaurant receives only 16 walk-ins in one night, then our solutions would be on target, while the others would only have accepted ~ 22 extra covers - which is only $\sim 72\%$ of the target (and of the performance by $SOLVER_1$ and $SOLVER_2$).

Figure 6.10 (bottom) zooms over the second half of the graphs. No significant difference is shown between $SOLVER_3$ and $SOLVER_4$. $SOLVER_2$ shows some improvement over $SOLVER_1$ in the second half of the graph. Ultimately, considering the entire dinner session and the number of extra covers after 30 walk-ins, $SOLVER_2$ is $\sim 1.5\%$ better than $SOLVER_1$, and $\sim 26\%$ better than the two solvers based on traditional methods.

Finally, Figure 6.11 shows the flexibility profile over time for the four solvers. Comparing the 4 versions of solver we can see how $SOLVER_1$ and especially $SOLVER_2$ maintain a higher flexibility compared to the others. The increase is partially due to the fact that our solutions do not fix parties to tables until the moment parties get seated, and partially to the optimization based on flexibility (for $SOLVER_2$). As the time passes the 4 curves decreases until they finally converge. Note how during floor management flexibility decreases much more quickly than during booking. Now, in fact, the chances to accommodate future requests decreases with the time, and so the flexibility is affected. For example, considering a dinner session from 4 p.m. to 10 p.m. as possible start times, then at 4 p.m. there are 6 hours (i.e. 25 15-minute time slots) available to start a meal, while at 10 p.m. there is only one last chance (or start time) available.



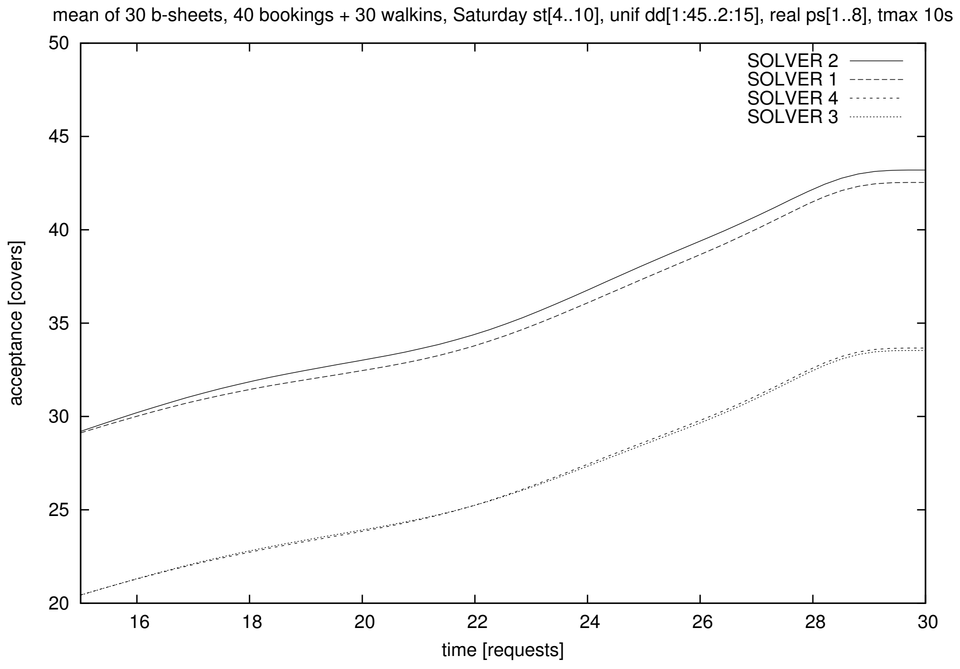


Figure 6.10: Tests on floor management: mean number of parties over number of walk-ins, comparing $SOLVER_1$, $SOLVER_2$, $SOLVER_3$, and $SOLVER_4$.

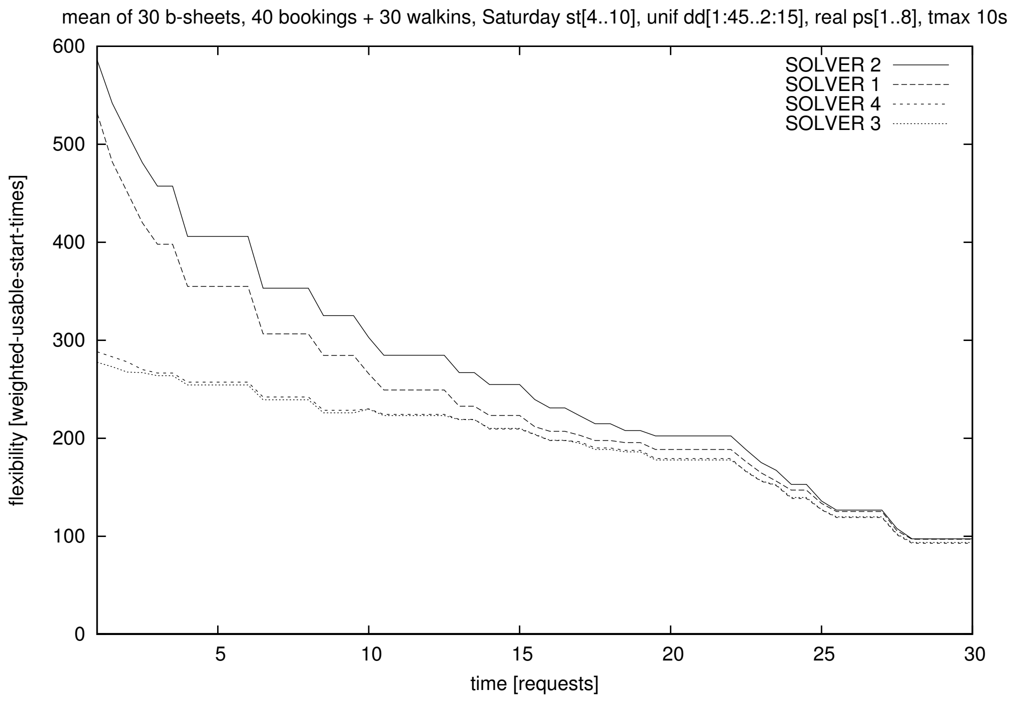


Figure 6.11: Tests on floor management: flexibility profile over number of walk-ins, comparing $SOLVER_1$, $SOLVER_2$, $SOLVER_3$, and $SOLVER_4$.

6.8 Floor management simulation on delays

We now test our system in terms of robustness to manage delays during floor management, i.e. the question we aim to address is:

- *can we accommodate a delay without delaying other parties?*

In particular, we want to evaluate the benefit coming from the fact that our system can automatically reconfigure the plan. The experiment we perform considers a set of M initial seating plans, each allocating N parties. The set of initial plans was generated in the following steps:[†]

i generate a scheduling problem S_i of N parties;

ii solve S_i, i.e. use our constraint satisfaction model (Chapter 4 and 5) to see whether S_i is feasible or not;

iii if S_i is feasible then record the plan SP_i provided by the model;

iv repeat (i), (ii), and (iii), until a set of M seating plans are collected.

The scheduling problems (and the seating plans) we generate have parties with start times uniformly distributed in the range 0 to 24 (i.e. 4 p.m. to 10 p.m. discretized in 15 minute units), and durations uniform in 6 to 10 (i.e. 90 to 150 minutes, again with a resolution of 15 minute units). The party size ranges between 1 and 8 with the %-frequency of each respectively 1, 43, 15, 14, 12, 9, 6, and 5.

6.8.1 Test on late finish

When the booker or the floor manager allocates parties to tables, they assign dinner slots by considering the standard dinner duration, which is typically around 2 hours. However, different customers have different behaviors, and even the same customer can stay longer or shorter depending on the occasion (e.g. business dinner, family dinner, anniversary, etc.). Dinner durations are thus uncertain. In this section we consider meals finishing later (rather than earlier) than expected, i.e.

[†]Note that we are not optimizing the initial seating plans.

we focus on those cases where duration uncertainty can disrupt a seating plan, and can cause delays for future dinners.

Given D, a delay in dinner duration (e.g. 15 minutes), the question we address in this test becomes:

> *when a current party of diners stays (D) longer than expected, can we maintain a feasible seating plan by reallocating future diners, without introducing other delays, and guaranteeing the same time slots currently allocated to the other parties?*

The time interval where delays can potentially cause disruptions (i.e. reallocations or further delays on future parties) is between 4 p.m. and 10 p.m., therefore we concentrate our test over this window. Specifically, for each set of parties representing our initial seating plans SP_i ($i = 1..M$), and for each party P_j ($j = 1..N$) in SP_i such that $P_j.end \leq$ 10 p.m., we perform the following test:

a fix the allocations in SP_i for time $t < P_j.end$;

b set $P_j.end = P_j.end + D$;

c resolve the scheduling problem S_i.

For each pair (SP_i, P_j) we record the following values:

- Nb of instances where the delay was accommodated with 0 changes;
- Nb of instances where the delay was accommodated with 1 change;
- Nb of instances where the delay was accommodated with ≥ 2 changes (with mean, std. deviation, and max number of changes);
- Nb of instances where the delay could not be accommodated (without delaying other parties).

6.8.2 Late finish with a restaurant load of 80%

We performed a first series of tests with $M = 30$ seating plans (to represent a month of simulation), and each plan with $N = 50$ parties (which provides a restaurant load of about 150 covers, i.e. 80% of the target of 180 covers). This gives total of 1500 parties over the 30 seating plans, while the number of parties with end time before 10 p.m. is 1023. Therefore we simulated 1023 instances of single delays. We repeated the experiments for different delays in dinner duration $D \in \{1, 2, 3, 4, 6, 12\}$, to represent delays of 15, 30, 45, 60, 90, and 180 minutes.

Results are reported in Table 6.4. The first column represents the delay in 15 minute units. The second is the number of instances (a fraction of the 1023) for which the delay was accommodated without changing the original plan. The third is the number for which the delay was accommodated by reallocating only a single party (i.e. the party following in the same table of the late finish). The fourth is the number for which multiple reallocations were necessary to accommodate the delay. The fifth and six are the mean (with standard deviation σ) and maximum number of changes performed over those instances requiring at least 2 reallocations. The final column shows the number of delay instances which were found infeasible, i.e. for which no reallocation was possible without delaying other parties. All the percentage values in the table refer to the total number of delay instances, i.e. 1023. The values of the second columns are particularly high. This is partially due to the fact that parties who are seated last in a table can finish late without any other party being delayed. Further, as the current restaurant load is 80%, the seating plans are not too dense, which is expected to increase the number of delays accommodated with minimum (i.e. 0 or 1) changes.

Current procedures would mostly cope with reallocations with zero or one changes, while instances which require multiple changes would be considered infeasible (as they are too hard to compute). So, at the moment, when a multiple reallocation without delays exists, the floor manager cannot find it, and therefore he has to delay future dinners and make people wait. From the results we can observe how our system can provide complex reallocations which absorb delays and maintain feasible seating plans in more cases. For instance, considering a delay of 15 minutes, current systems would have to introduce delays in

$14.0\% + 2.5\% = 16.5\%$ of the cases, while using our system we would only need to delay some future dinners in 2.5% of the cases. For a delay of 15 minutes, and considering only instances requiring two or more changes, the mean number of changes was 3.3 (with standard deviation 4.1), while the maximum number for a single instance was 30. These values give us an index of the complexity of the operations that were performed.

Table 6.4: Late finishes. Results over 30 seating plans of 50 parties - i.e. for each delay value D we test 1023 delay instances.

Delay D	**Nb 0 CHs**	**Nb 1 CHs**	**Nb 2^+ CHs**	**mean (σ)**	**max**	**Nb infeasibles**
1 (15min)	630 61.6%	224 21.9%	143 14.0%	3.3 (4.1)	30	26 2.5%
2 (30min)	400 39.1%	373 36.5%	207 20.2%	3.8 (5.1)	33	43 4.2%
3 (45min)	269 26.3%	460 45.0%	233 22.8%	4.0 (5.6)	29	61 6.0%
4 (60min)	269 26.3%	459 44.9%	223 21.8%	4.2 (5.5)	29	72 7.0%
6 (90min)	269 26.3%	459 44.9%	212 20.7%	4.6 (5.9)	34	83 8.1%
12 (120min)	190 18.6%	361 35.3%	365 35.7%	4.7 (6.3)	33	107 10.5%

As we increase the delay, we can observe that the percentage of instances requiring 0 changes significantly decreases, then stabilizes to 26.3%, and finally goes down to 18.6% for $D = 120$ minutes. This very long delay is perhaps not realistic, but the 18.6% of instances solved with 0 changes gives us an estimate of the percentage of parties (with end time before 10 p.m.) who are seated last in a table, i.e. who can always be late finish without delaying other parties. The percentage of instances requiring 1 and those requiring 2 or more changes increase as the delay goes from 15 to 30 to 45 minutes, and they get stable for delays of 60 and 90 minutes. Note how the final delay of 120 minutes makes the instances requiring 1 change decrease (to 35.3%), while those requiring multiple reallocations significantly increase (to 35.7%). For the latter instances, the mean number of changes also rises from 3.3 to 4.7 (and the deviation from 4.1 to 6.3) as the delay value moves from 15 to 120 minutes. The maximum number of changes is always around 30. Finally, as expected, the number of infeasible instances gets larger for larger delays.

6.8.3 Late finish with a restaurant load of 100%

We performed a second round of tests, still with $M = 30$ seating plans, but now with $N = 60$ parties (which provides a restaurant load of approximately 180 covers, i.e. 100% of the restaurant turnover target). This gives total of 1800 parties over the 30 seating plans, while the number of parties with end time before 10 p.m. is 1267. Therefore we simulated 1267 instances of single delays. Similarly to the previous test, we repeated the experiments with D = 1, 2, 3, 4, 6, and 12 (15 minute units).

Results are reported in Table 6.5. Considering a delay of 15 minutes, current systems would have to introduce delays in $12.4\% + 4.6\% = 17\%$ of the cases, while with our system we would need to delay some future dinners in only 4.6% of the cases. For a delay of 15 minutes, and considering only instances requiring two or more changes, the mean number of changes was 3.5 (with standard deviation 4.8), while the maximum number for a single instance was 31. This again indicates how complex are the operations which were performed.

Table 6.5: Late finishes. Results over 30 seating plans of 60 parties - i.e. for each delay value D we test 1267 delay instances.

Delay D	**Nb 0 CHs**	**Nb 1 CHs**	**Nb 2^+ CHs**	**mean (σ)**	**max**	**Nb infeasibles**
1	733	319	157	3.5	31	58
(15min)	57.9%	25.2%	12.4%	(4.8)		4.6%
2	487	464	212	3.8	32	104
(30min)	38.4%	36.6%	16.7%	(5.2)		8.2%
3	366	527	232	4.2	32	142
(45min)	28.9%	41.6%	18.3%	(5.6)		11.2%
4	273	579	251	4.8	36	164
(60min)	21.5%	45.7%	19.8%	(7.1)		12.9%
6	200	611	248	6.3	44	208
(90min)	15.8%	48.2%	19.6%	(8.4)		16.4%
12	136	374	476	5.6	48	281
(120min)	10.7%	29.5%	37.6%	(8.4)		22.2%

The values of the second column (instances solved with 0 reallocations) decreases monotonically as the delay increases, and generally, compared Table 6.4 the numbers are now lower. This can be explained considering that the new seating plans are more dense (180 rather than 150 covers on average), which make the restaurant less flexible to accept delays with minimum changes. Further, with a delay of 120 minutes the instances with 0 changes decreases from 18.6% to 10.7% - now the number of parties is 60, so for the same number of tables (23) the percentage of parties seating last in a table is smaller. The percentage of instances requiring 1 change increases up to 48.2% as the delay goes from 15 to 90 minutes, but the number drops to 29.5% for a delay of 120 minutes. The instances requiring multiple changes increase almost monotonically over the 5 steps of delays, eventually getting to 37.6%. This tells us how the complexity of the operations gets harder when delays increase, and especially when the seating plans get crowded. For instances requiring multiple changes, the mean number of changes rises from 3.5 to 5.6 (and the deviation from 4.8 to 8.4) for increasing delay values. Similarly, the maximum number of changes goes from 31 to 48. Notice how the mean and maximum values are greater compared to those of Table 6.4 - in fact, seating plans have now 60 rather than 50 parties, which allows more reallocations. Finally, the number of infeasible instances is also higher with instances of 60 rather than 50 parties. Again, as expected, the number of infeasibles increases as the delay gets larger.

6.8.4 Test on late arrivals

Often, customers arrive late without giving any notice - a typical delay can be anything up to 30 minutes, after which the manager would normally consider the booking a no-show. Other times, instead, customers phone up advising the restaurant of the late arrival - the delay can be 15 or 30 minutes, but in some cases also over one hour. Further, another regular occurrence is when the customer and the booker have recorded different times for the booking.

Given D, a delay in party arrival (e.g. 15 minutes), the question we address in this test is:

> *when a party arrives (D) later than expected, can we maintain a feasible seating plan by reallocating future diners, without introducing other delays, and guaranteeing the same time slots currently allocated to the other parties?*

Similarly to what we have done for the previous case on delays in dinner duration, for each seating plan SP_i ($i = 1..M$), and for each party P_j ($j = 1..N$) in SP_i such that $P_j.end \leq 10$ p.m., we now perform the following test:

a fix the allocations in SP_i for time $t < P_j.start$;

b set $(P_j.start, P_j.end) = (P_j.start + D, P_j.end + D)$;

c resolve the scheduling problem S_i.

Note that, the time interval where a late arrival can cause disruptions is different (i.e. earlier) compared to the case of a late finish: with a dinner session where customers get seated between 4 p.m. and 10 p.m., and with a minimum dinner duration of 1:30 hours, disruptions can be caused by late arrivals in the range 4:00 p.m. to 8:30 p.m., and by late finishes in the range 5:30 p.m. to 10:00 p.m.. Then, generally, a late arrival occurs at an earlier time compared to a late finish, so the time window where parties can be reallocated is larger.

6.8.5 Late arrival with restaurant loads of 80% and 100%

We again performed two rounds of tests, both with $M = 30$ seating plans, one with $N = 50$ parties and one with $N = 60$ parties (which provides a restaurant load of about 150 and 180 covers in the order, i.e. 80% and 100% of the restaurant turnover target). We repeated the test for delays $D \in \{1, 2, 4, 6\}$ 15 minutes units, to represent parties arriving late by 15, 30, 60, and 90 minutes. Table 6.6 and Table 6.7 report the results. The values (and percentages) in the tables are relative to the total number of delay instances, i.e. 1023 for the test with lower restaurant load, and 1267 for the test with higher load - we used the same two sets of initial seating plans generated for the previous tests on late finish.

Table 6.6: Late arrivals. Results over 30 seating plans of 50 parties - i.e. for each delay value D we test 1023 delay instances.

Delay D	**Nb** 0 **CHs**	**Nb** 1 **CHs**	**Nb** 2^+ **CHs**	**mean** (σ)	**max**	**Nb infeasibles**
1 (15min)	496 48.5%	286 28.0%	218 21.3%	9.07 (14.5)	49	23 2.2%
2 (30min)	269 28.5%	413 40.4%	309 30.2%	8.23 (14.4)	50	32 3.1%
4 (60min)	177 17.3%	479 46.8%	290 28.3%	12.92 (18.8)	48	77 7.5%
6 (90min)	198 19.4%	467 45.6%	275 26.9%	12.63 (16.3)	48	83 8.1%

Table 6.7: Late arrivals. Results over 30 seating plans of 60 parties - i.e. for each delay value D we test 1267 delay instances.

Delay D	**Nb** 0 **CHs**	**Nb** 1 **CHs**	**Nb** 2^+ **CHs**	**mean** (σ)	**max**	**Nb infeasibles**
1 (15min)	732 57.8%	396 31.3%	98 7.7%	7.50 (13.3)	55	41 3.2%
2 (30min)	486 38.4%	566 44.7%	145 11.4%	7.42 (12.8)	55	70 5.5%
4 (60min)	272 21.5%	671 53.0%	218 17.2%	8.59 (14.2)	57	106 8.4%
6 (90min)	200 15.8%	668 52.7%	235 18.5%	10.97 (15.8)	59	164 12.9%

As expected, the infeasible instances increase for increasing delays, and the number is greater for more crowded seating plans, i.e. passing from 50 parties (Table 6.6) to 60 parties (Table 6.7). For seating plans with 50 parties, the percentage of instances requiring two or more changes is quite high even for a short delay of 15 minutes (21.3%), and it goes up to approximately 27-30% for longer delays. For example, for a delay of 15 minutes, current systems would consider 21.3% + 2.2% = 23.5% of late arrival instances as infeasible, for which the floor manager would need to delay some future dinners in order to fit everybody in. Using our system, instead, the restaurant would need to introduce delays only in 2.2% of the cases. For the remaining 21.3%, in fact, delays are not necessary, as our system can find feasible reallocations which accommodate the late arrivals while preserving the original start times and durations of all the other meals. For seating plans with 60 parties, the percentage of instances requiring two or more changes is generally lower but still relevant, going from 7.7% for a delay of 15 minutes to 18.5% for a delay of 90 minutes.

The mean, standard deviation, and maximum number of changes required for the accommodation of late arrivals give us an indication of the complexity of the reallocations performed by the system. Table 6.6 shows a mean from 8.23 to 12.63 changes and deviation from 14.4 to 18.8 (column 5), and a maximum from 48 to 50 (column 6), while Table 6.7 shows a mean from 7.42 to 10.97 changes, a deviation from 7.42 to 10.97, and a maximum from 55 to 59. Note that, for any restaurant load (80% or 100%) and delay ($D \in \{1, 2, 4, 6\}$), the mean, deviation, and max number of changes in the case of instances requiring multiple reallocations are significantly higher compared to the results discussed in the previous experiments for delays in dinner durations. This is explained considering that the fraction of parties which can be reallocated is now generally larger. In fact, for example, a late arrival can happen at time 4:00 p.m., in which case all parties, starting in the entire time window from 4:00 p.m. to 10:00 p.m., are free for reallocation. Instead, considering a standard dinner duration of 2 hours, a late finish can happen at time 6:00 (at the earliest), in which case parties are free for reallocation only in the time window from 6:00 p.m. to 10:00 p.m., i.e. parties starting before 6:00 p.m. are already seated (and therefore fixed). Thus, on average, a delay in dinner duration allows fewer options to reallocate future parties. Note, instead, how the maximum number of changes for a delay in arrival time reaches approximately 50 in Table 6.6 and approximately 60 in Table 6.7, i.e. the system can effectively reallocate entire seating plans to accommodate late arrivals.

Finally, with 50 parties (Table 6.6), for values of delay from $D = 15$ to 30 to 60 minutes, the percentage of instances requiring 0 changes decreases from 48.5% to 17.3% (re-increasing to 19.4% for $D = 90$ minutes), while the number requiring 1 change increases from 28.0% to 46.8% (re-decreasing to 45.6% for $D = 90$ minutes). With 60 parties (Table 6.7), for increasing delays the percentage of instances requiring 0 changes decreases monotonically from 57.8% to 15.8%, while the number requiring 1 change increases almost monotonically from 31.3% to 52.7%. As already pointed out for the case of delays in dinner duration, note that all parties who are allocated last in a table can be delayed without modifying the original seating plan. This explains the fact that the percentage of late arrivals requiring zero changes (second columns) can be higher than expected even for a crowded restaurant.

6.9 Chapter summary

In this chapter we have shown how our constraint based techniques can cope with the dynamic problem. We first introduced the dynamic problem for booking and floor management based on scheduling. We then presented our allocation methods (for satisfaction and optimization) and implemented also two versions simulating traditional methods currently adopted in restaurants. We extended our search algorithm to consider solution (or seating plan) stability. We finally presented the experimental results, showing how our solutions outperform the traditional allocation systems, on both booking and floor simulations.

A first contribution to the ability to solve the dynamic problem comes from the fact that our solutions got rid of the (unnecessary) commitment of parties to tables before the actual seating time. Results showed that without such commitment there is a consistent improvement in the number of customers the restaurant can accept, and, in terms of robustness to absorb delays, we consistently reduce the cases where delays (either late finishes or late arrivals) require some future dinners to be delayed.

A second contribution concerns our flexibility based optimization, i.e. the fact that the solutions we provide are designed to be flexible for the future. Results showed there is a further improvement by performing optimization online, after each change. Optimization makes a difference when we are asked to decide where to fix some parties (i.e. during the floor phase, when parties get seated over time). Further, as our algorithm searches first in the neighbourhood of the preceding seating plan, results on the booking phase (i.e. when no allocation is fixed) suggest that the search for the accommodation of a new request takes advantage if the preceding plan is more flexible to accept future requests with minimum changes.

A further test was carried out to investigate the potential benefit of using optimization to exploit customer's start time flexibility, i.e. when some customers are flexible over the time to consume their dinner. Results showed that there is a significant improvement in the number of customers the restaurant can accommodate if optimization is used to guide start time negotiation.

We then evaluated and compared three versions of our optimization model based on three flexibility measures, i.e. maximizing usable start times, minimizing dead zones, and maximizing the number of potential seatings. Results showed that the three models have very similar performances, and that the three measures are all reasonably accurate in representing the real flexibility.

Ultimately, we evaluated the ability of our model to represent and make use of future knowledge. We performed a final experiment with parties distributed according to a pattern for a Sunday session at the Eco restaurant. Results showed that the improvement coming from using the correct pattern is negligible if we assume all customers have no flexibility over start times. With some flexibility over start times, instead, the benefit of using a correct booking pattern becomes clearly noticeable - i.e. our model of future knowledge becomes more effective when we can select the most flexible time among different alternatives.

Chapter 7

Restaurant trials

7.1 Introduction

In the previous chapters, we first described the problem of restaurant table management, we then presented our constraint satisfaction and optimization models of the problem (based on scheduling), and finally, we tested our solutions using computer simulations. In this chapter, we complete the evaluation of our solutions, presenting the results of a six month trial in the restaurant Eco. In the first part, we describe the software prototype which was used by the restaurant management and staff during the trial, focusing on the main features which have been implemented, and on the different types of advice they can provide in real time.[†] In the second part, we present the outcome of the evaluation, discussing a questionnaire which was completed out by the general manager of the restaurant.

The evaluation in the restaurant Eco was carried out to verify the results achieved in simulation, and in general, to assess the validity of our research in the real environment. Specifically, the aim of the trial was to see whether the software:

- models the restaurant adequately;
- provides acceptable/flexible seating plans in reasonable time;

[†]All features described in this chapter are provided by the constraint models presented in the previous chapters. The GUI that was used to allow the restaurant staff to interact with the models was specified by the author, but was implemented by James Lupton. GUI examples are used throughout this chapter to make the presentation clearer.

- can join and separate tables correctly;
- reports quickly whether or not a request can be accepted;
- recommends sensible alternative times for a booking;
- provides useful advice when a seating plan has to be reconfigured.

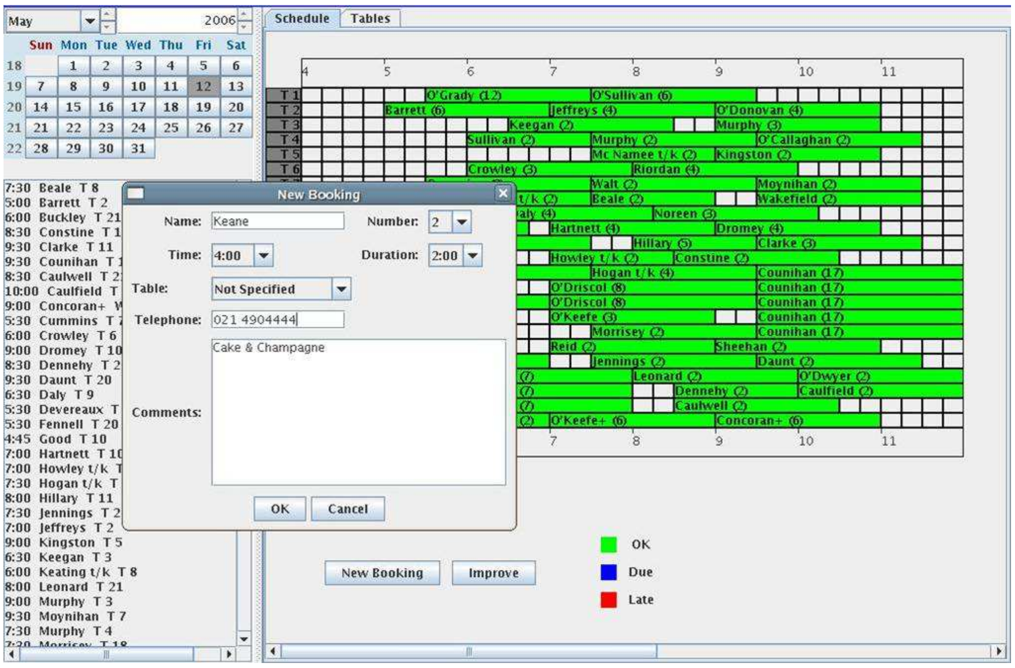
7.2 The prototype

The models and algorithms described in the previous chapters 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, optimizing for flexibility, or maintaining stability.

A screen shot of the interface is shown in Figure 7.1, 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.

7.2.1 Adding a new request

Figure 7.2 represents the seating plan accommodating the new request (*Keane*). The solution was returned in 0.04 seconds. It also shows the total covers, the covers partitioned in three 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. The user can switch from the schedule view to a table map, shown in Figure 7.4.



7:30 Beale T 8
5:00 Barrett T 2
6:00 Buckley T 21
8:30 Constine T 1
9:30 Clarke T 11
9:30 Counihan T 1
8:30 Caulwell T 2
10:00 Caulfield T
9:00 Concoran+ W
5:30 Cummins T 7
6:00 Crowley T 6
9:00 Dromey T 10
8:30 Dennehy T 2
9:30 Daunt T 20
6:30 Daly T 9
5:30 Devereaux T
5:30 Fennell T 20
4:45 Good T 10
7:00 Hartnett T 10
7:00 Howley t/k T
7:30 Hogan t/k T
8:00 Hillary T 11
7:30 Jennings T 2
7:00 Jeffreys T 2
9:00 Kingston T 5
6:30 Keegan T 3
6:00 Keating t/k T 8
8:00 Leonard T 21
9:00 Murphy T 3
9:30 Moynihan T 7
7:30 Murphy T 4

Name:	Keane	Number:	2
Time:	4:00	Duration:	2:00
Table:	Not Specified		
Telephone:	021 4904444		
Comments:	Cake & Champagne		

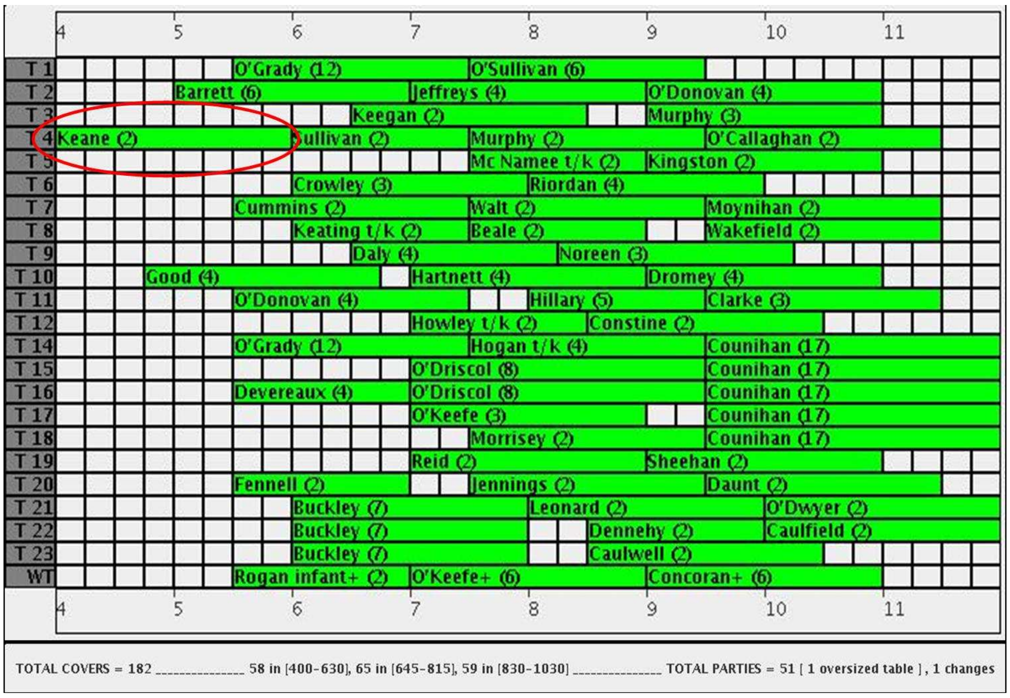
Figure 7.1: User interface, displaying a seating plan and a new booking request.

7.2.2 Making a preference

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 Figure 7.3, party *Keane* has been moved to table 11, which is for 5 people. The operation required 3 changes from the previous table allocation, and was performed in 0.23 seconds. The names of parties with preferences are displayed with a suffix "+".

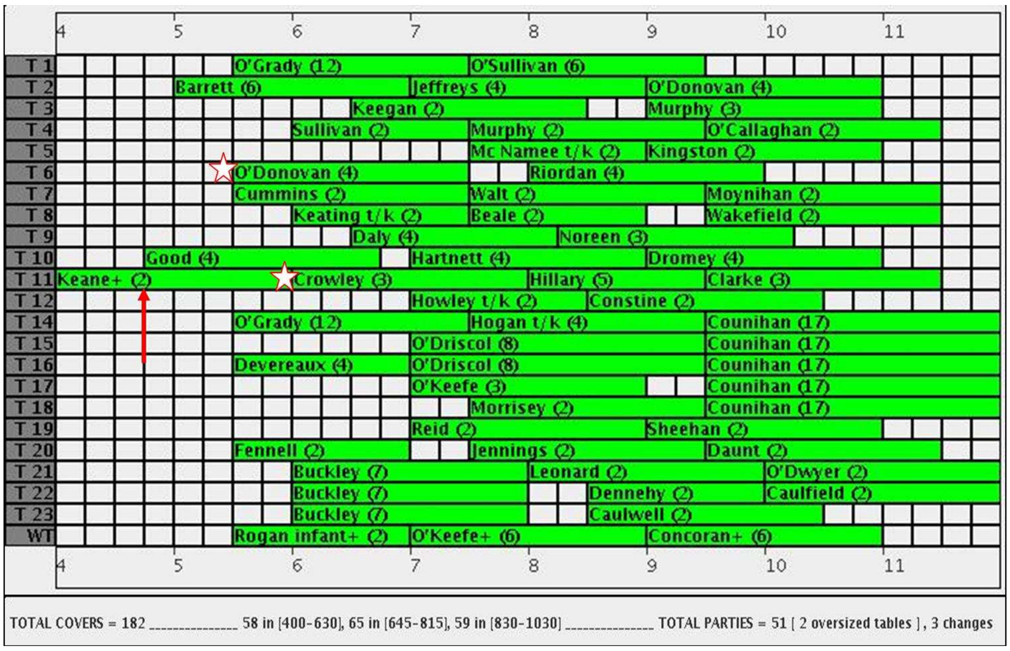
7.2.3 Availability enquiry

During booking, availability requests are common - e.g. "when can you seat a party of 4?". The user can process such request using the same booking form, by selecting "not specified" in the *Time* box. Figure 7.6 shows the current seating plan (top) and the answer provided by the system for a request for 4 people (bottom). The message also groups the available times by the available duration. This is important information, since the booker may be able to sell the table for one hour


4 5 6 7 8 9 10 11
T 1 O'Grady (12) O'Sullivan (6)
T 2 Barrett (6) Jeffreys (4) O'Donovan (4)
T 3 Keegan (2) Murphy (3)
T 4 Keane (2) Sullivan (2) Murphy (2) O'Callaghan (2)
T 5 Mc Namee t/k (2) Kingston (2)
T 6 Crowley (3) Riordan (4)
T 7 Cummins (2) Walt (2) Moynihan (2)
T 8 Keating t/k (2) Beale (2) Wakefield (2)
T 9 Daly (4) Noreen (3)
T 10 Good (4) Hartnett (4) Dromey (4)
T 11 O'Donovan (4) Hillary (5) Clarke (3)
T 12 Howley t/k (2) Constine (2)
T 14 O'Grady (12) Hogan t/k (4) Counihan (17)
T 15 O'Driscol (8) Counihan (17)
T 16 Devereaux (4) O'Driscol (8) Counihan (17)
T 17 O'Keefe (3) Counihan (17)
T 18 Morrisey (2) Counihan (17)
T 19 Reid (2) Sheehan (2)
T 20 Fennell (2) Jennings (2) Daunt (2)
T 21 Buckley (7) Leonard (2) O'Dwyer (2)
T 22 Buckley (7) Dennehy (2) Caulfield (2)
T 23 Buckley (7) Caulwell (2)
WT Rogan infant+ (2) O'Keefe+ (6) Concoran+ (6)

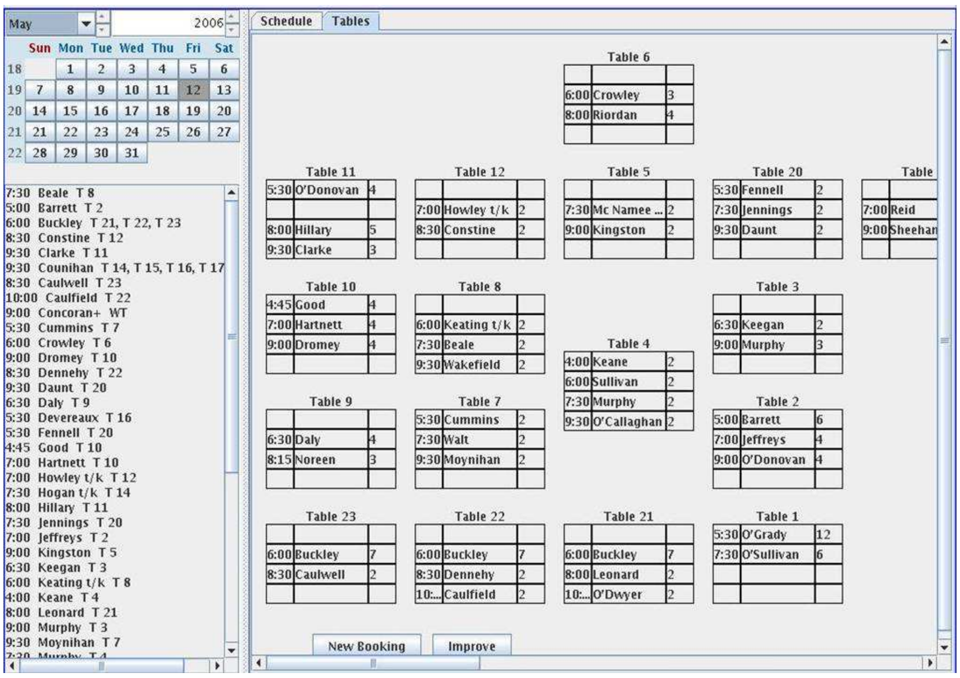
TOTAL COVERS = 182 _______________ 58 in [400-630], 65 in [645-815], 59 in [830-1030] _______________ TOTAL PARTIES = 51 [1 oversized table], 1 changes

Figure 7.2: Seating plan with the new request accommodated into table 4.


4 5 6 7 8 9 10 11
T 1 O'Grady (12) O'Sullivan (6)
T 2 Barrett (6) Jeffreys (4) O'Donovan (4)
T 3 Keegan (2) Murphy (3)
T 4 Sullivan (2) Murphy (2) O'Callaghan (2)
T 5 Mc Namee t/k (2) Kingston (2)
T 6 O'Donovan (4) Riordan (4)
T 7 Cummins (2) Walt (2) Moynihan (2)
T 8 Keating t/k (2) Beale (2) Wakefield (2)
T 9 Daly (4) Noreen (3)
T 10 Good (4) Hartnett (4) Dromey (4)
T 11 Keane+ (2) Crowley (3) Hillary (5) Clarke (3)
T 12 Howley t/k (2) Constine (2)
T 14 O'Grady (12) Hogan t/k (4) Counihan (17)
T 15 O'Driscol (8) Counihan (17)
T 16 Devereaux (4) O'Driscol (8) Counihan (17)
T 17 O'Keefe (3) Counihan (17)
T 18 Morrisey (2) Counihan (17)
T 19 Reid (2) Sheehan (2)
T 20 Fennell (2) Jennings (2) Daunt (2)
T 21 Buckley (7) Leonard (2) O'Dwyer (2)
T 22 Buckley (7) Dennehy (2) Caulfield (2)
T 23 Buckley (7) Caulwell (2)
WT Rogan infant+ (2) O'Keefe+ (6) Concoran+ (6)

TOTAL COVERS = 182 _______________ 58 in [400-630], 65 in [645-815], 59 in [830-1030] _______________ TOTAL PARTIES = 51 [2 oversized tables], 3 changes

Figure 7.3: New seating plan after imposing a preference for party *Keane*.



```
7:30 Beale T 8
5:00 Barrett T 2
6:00 Buckley T 21, T 22, T 23
8:30 Constine T 12
9:30 Clarke T 11
9:30 Counihan T 14, T 15, T 16, T 17
8:30 Caulwell T 23
10:00 Caulfield T 22
9:00 Concoran+ WT
5:30 Cummins T 7
6:00 Crowley T 6
9:00 Dromey T 10
8:30 Dennehy T 22
9:30 Daunt T 20
6:30 Daly T 9
5:30 Devereaux T 16
5:30 Fennell T 20
4:45 Good T 10
7:00 Hartnett T 10
7:00 Howley t/k T 12
7:30 Hogan t/k T 14
8:00 Hillary T 11
7:30 Jennings T 20
7:00 Jeffreys T 2
9:00 Kingston T 5
6:30 Keegan T 3
6:00 Keating t/k T 8
4:00 Keane T 4
8:00 Leonard T 21
9:00 Murphy T 3
9:30 Moynihan T 7
```

Table 6

6:00	Crowley	3
8:00	Riordan	4

Table 11

5:30	O'Donovan	4
8:00	Hillary	5
9:30	Clarke	3

Table 12

7:00	Howley t/k	2
8:30	Constine	2

Table 5

7:30	Mc Namee ...	2
9:00	Kingston	2

Table 20

5:30	Fennell	2
7:30	Jennings	2
9:30	Daunt	2

Table

7:00	Reid
9:00	Sheehan

Table 10

4:45	Good	4
7:00	Hartnett	4
9:00	Dromey	4

Table 8

6:00	Keating t/k	2
7:30	Beale	2
9:30	Wakefield	2

Table 3

6:30	Keegan	2
9:00	Murphy	3

Table 4

4:00	Keane	2
6:00	Sullivan	2
7:30	Murphy	2
9:30	O'Callaghan	2

Table 9

6:30	Daly	4
8:15	Noreen	3

Table 7

5:30	Cummins	2
7:30	Walt	2
9:30	Moynihan	2

Table 2

5:00	Barrett	6
7:00	Jeffreys	4
9:00	O'Donovan	4

Table 23

6:00	Buckley	7
8:30	Caulwell	2

Table 22

6:00	Buckley	7
8:30	Dennehy	2
10:...	Caulfield	2

Table 21

6:00	Buckley	7
8:00	Leonard	2
10:...	O'Dwyer	2

Table 1

5:30	O'Grady	12
7:30	O'Sullivan	6

Figure 7.4: User interface, displaying a table map of the new seating plan with party *Keane*.

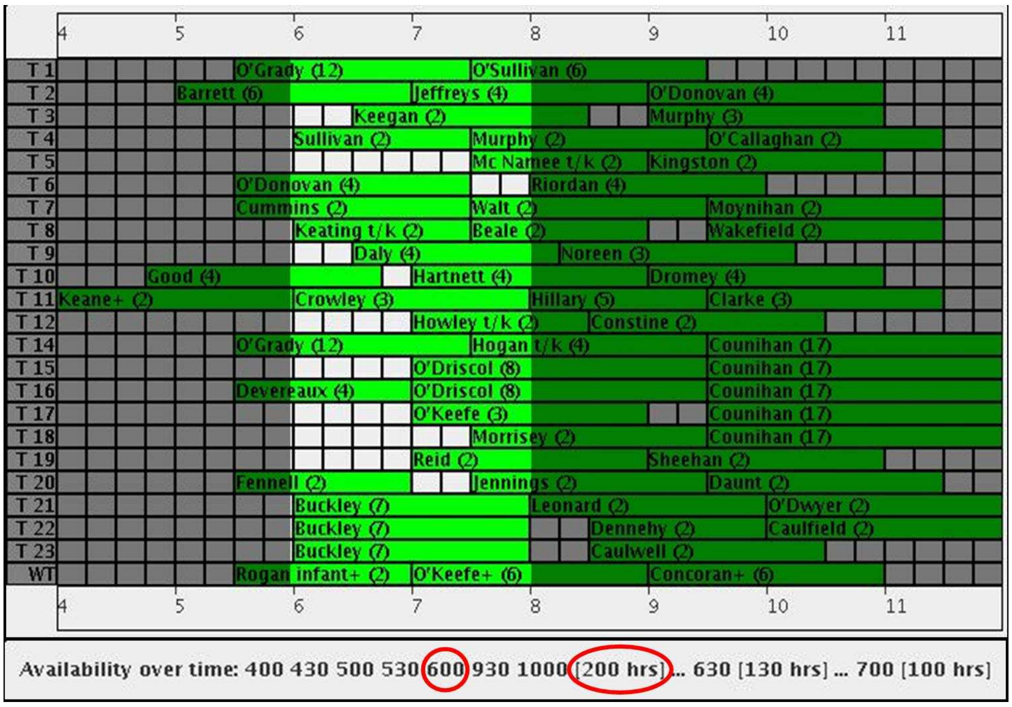
at 7 o'clock if the customer is only asking for a quick main course.

The pseudocode for the procedure which provides this information is shown in Figure 7.5. The list of parties in the seating plan is represented by $\{P_1, P_2, .., P_n\}$, while P_{n+1} is added to compute the availability request. The procedure checks the availability for a dinner of the specified size (*new request's size*), resolving the problem for each possible start time - it starts by assigning the standard dinner duration (e.g. 2 hours for a party of four), and repeats the search with reduced durations for those start times where the standard dinner length cannot be accommodated. We set a timeout of 1 second for each combination of start time and duration, which allows us to obtain the availability message in a reasonable time (less than 10 seconds on average). Note that a dinner slot of two hours is reported to be available at 6 o'clock, even though in the current seating plan no table can accommodate 4 people for two hours at that time - i.e. our system guarantees that there is an alternative seating plan with such a table slot available.

$$CSP(X, D, C) : X = \{P_1, P_2, .., P_n\} \cup P_{n+1},$$
$$P_{n+1} \;=\; \langle size_{n+1},\ start_{n+1},\ duration_{n+1} \rangle$$

```
$size_{n+1} \;=\; new\,request's\,size;$
for $start_{n+1} \;=\; 4:00\,p.m.\ \ldots\ 10:00\,p.m.$ (step $30'$)
    for $duration_{n+1} \;=\; stdDuration\ \ldots\ stdDuration - 1hour$ (step $30'$)
        if  $Solve(CSP,\ timeout) == $ true
            $AVAILABILITY\ [size_{n+1}]\ [start_{n+1}]\ [duration_{n+1}] = 1;$
            break;
```

Figure 7.5: Procedure for computing availability by start time and duration.



Availability over time: 400 430 500 530 600 930 1000 [200 hrs] ... 630 [130 hrs] ... 700 [100 hrs]

Figure 7.6: Computing the possible start times (and durations) available for a dinner for 4 people.

7.2.4 Adding a new request which cannot fit in the current plan

Figure 7.7 (top) represents the current seating plan with a new request (party *Meane*, 2 people, 9 o'clock), and Figure 7.7 (bottom) shows a reallocation that the system found in less than 5 seconds - the new plan accommodates the party *Meane* in table 8, which is a two seater. Note that the party could not fit in the original seating plan. The system performed a complex operation, as the number of changes necessary to find the new plan was 43, with only 9 parties (highlighted with white stars) preserving the original table.

7.2.5 Improving seating plan flexibility

Figure 7.8 (top) highlights how the current seating plan is perhaps poor in quality. We can observe a party of size 3 (*Crowley*) accommodated in a table for 6 ($T6$), and there are several time intervals (cross hatched squares) which are dead-zones, i.e. tables are idle but the intervals are too short to accommodate future requests. Figure 7.8 (bottom) represents a first step in a search for a more flexible allocation. The new plan has been obtained pressing the *Improve* button (Figure 7.1). Note that there has been only one change from the previous plan, with party *Crowley* moved from table 6 (6-seater) to table 9 (4-seater). Based on flexibility$_{WUS}$ (defined in Chapter 5), and on a standard dinner duration of 2 hours (or 8 units of 15 minutes), the increase in the flexibility estimate is 16, i.e. 8 time units $\times$ 2 table size saved. This may allow an extra 2-hour dinner (8 time units) for two people. In fact, note that the new seating plan could accept a party of size 6 at 6 o'clock (in table 6), while the old plan could only accept a party of size 4 at 6 o'clock (in table 9). The run time to obtain the change is 0.16 sec.

The user can repeat the improvement process to find more flexible seating plans. Figure 7.9 (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 steps, 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 zones (empty squares) in between parties. The increase in the flexibility estimate over Figure 7.8 (top) is 68 and 96 for Figure 7.9 (top)

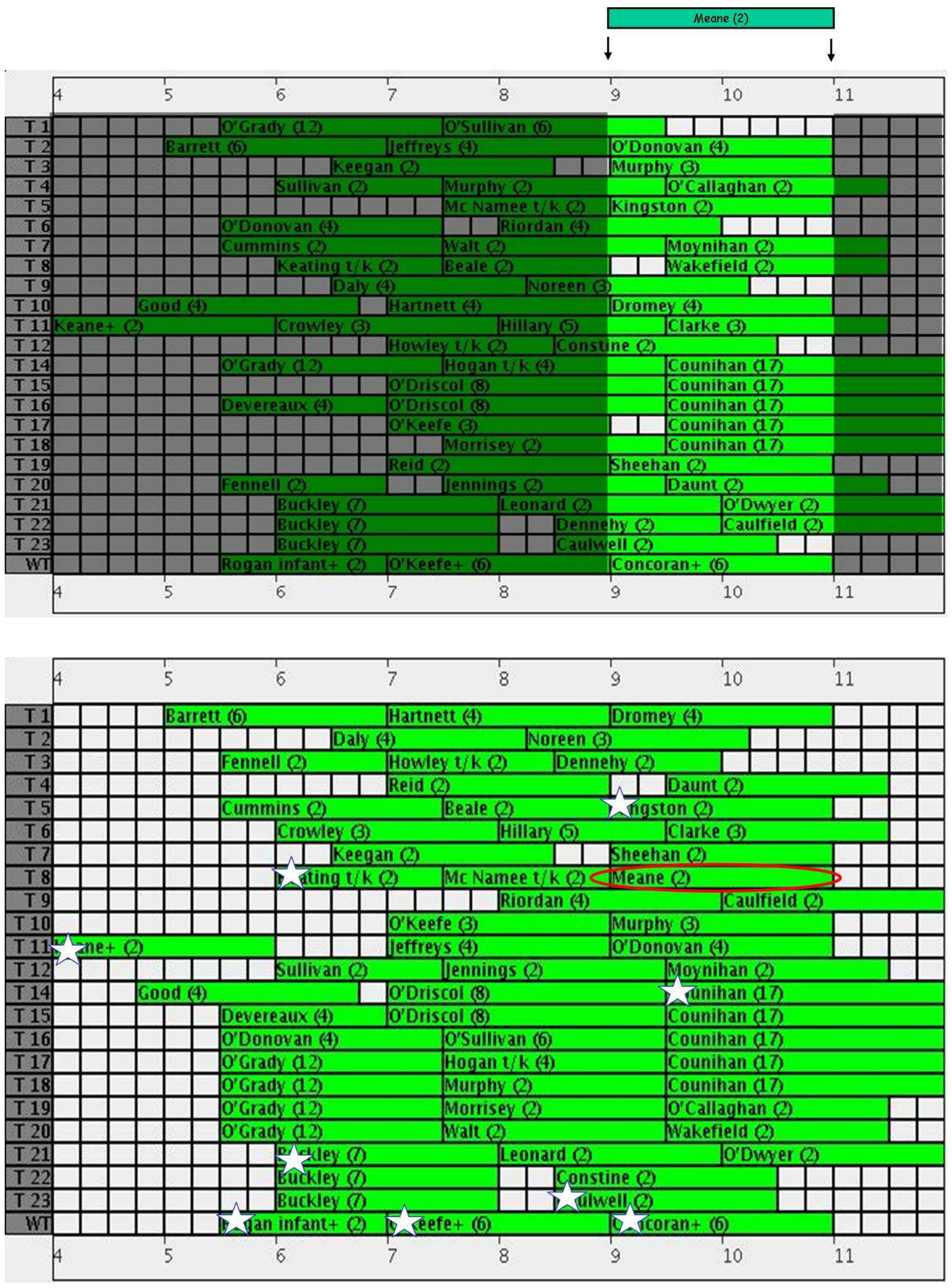


Figure 7.7: New seating plan with party *Meane* accommodated.

and (bottom), in the order. This can be regarded as 3 and 5.5 times the (2 hour $\times$ 2 people) improvement obtained from the first step of Figure 7.8 (bottom). In particular, note how the number of unusable squares (dead-zones) in the final plan is only 3 - there were 13 in the initial plan of Figure 7.8 (top). The run time from Figure 7.8 (bottom) to Figure 7.9 (top) was 8.1 seconds, and from Figure 7.9 (top) to (bottom) was 1.01 seconds.

7.2.6 Providing availability by flexibility

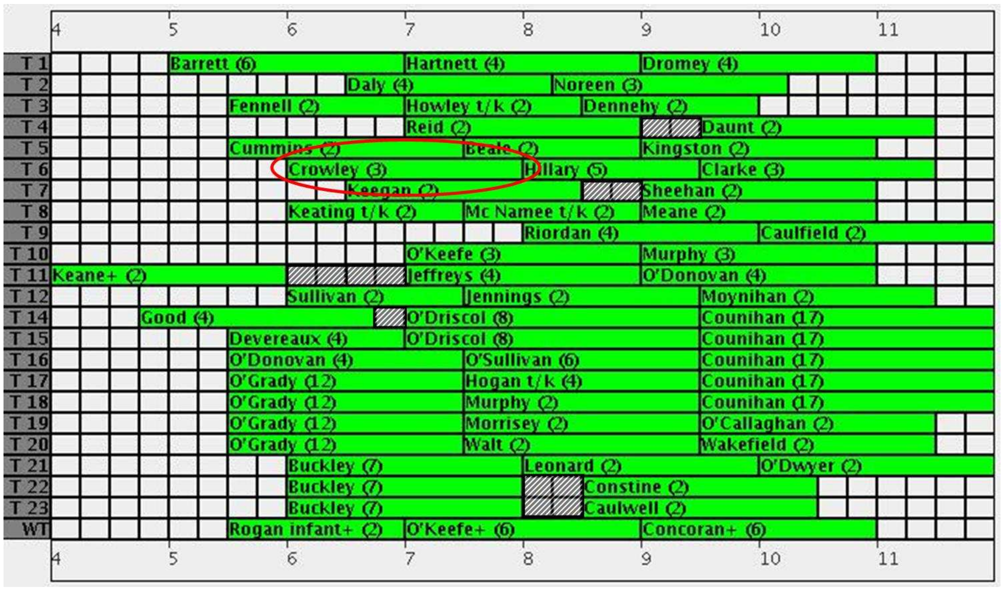
When a booking request cannot be satisfied because the restaurant is fully booked for the requested time, or when the customer is flexible over the time to consume his dinner, the system can be used to calculate what are the remaining dinner slots available (as discussed in Figure 7.5 and Figure 7.6). The system can actually help the booker to negotiate more flexible time slots by measuring (and displaying) the flexibility associated to each available start time and dinner duration.

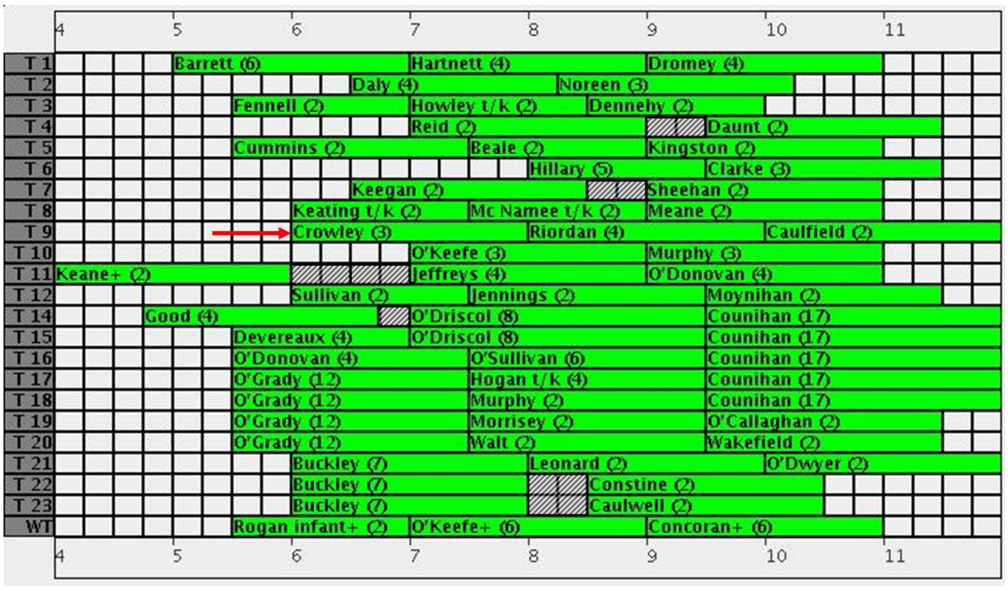
The procedure in Figure 7.10 computes the available dinner slots (*start time*, *duration*) for a given *new request's size*, as in Figure 7.5. However, each time a dinner slot is found available, the system now records the correspondent seating plan (including the new "virtual" request) and spends an interval of time timeout$_2$ to look for improved seating plans. Each time an improvement is found, the seating plan is updated and so is the flexibility value. At the end of timeout$_2$ the value of flexibility associated to the current party size, start time, and duration represents our (best) estimate of flexibility for the correspondent dinner slot.

Figure 7.11 (top) is a message returned by the application, showing the flexibility of the available dinner slots for a booking request for 4 people for the seating plan of Figure 7.9 (top). The values of flexibility are computed using our measure based on usable start times (i.e. flexibility$_{WUS}$). The messages can be read as in the following example:

$$\langle\ start_1\_flex_1\ start_2\_flex_2\ start_3\_flex_3\ [dur_1]...start_4\_flex_4\ [dur_2]...start_5\_flex_5\ [dur_3]\ \rangle.$$

Figure 7.11 (bottom) is the correspondent message for the plan of Figure 7.9 (bottom). Note how the second message reports higher flexibility values for every


4 5 6 7 8 9 10 11
T1 Barrett (6) Hartnett (4) Dromey (4)
T2 Daly (4) Noreen (3)
T3 Fennell (2) Howley t/k (2) Dennehy (2)
T4 Reid (2) Daunt (2)
T5 Cummins (2) Beale (2) Kingston (2)
T6 Crowley (3) Hillary (5) Clarke (3)
T7 Keegan (2) Sheehan (2)
T8 Keating t/k (2) Mc Namee t/k (2) Meane (2)
T9 Riordan (4) Caulfield (2)
T10 O'Keefe (3) Murphy (3)
T11 Keane+ (2) Jeffreys (4) O'Donovan (4)
T12 Sullivan (2) Jennings (2) Moynihan (2)
T14 Good (4) O'Driscol (8) Counihan (17)
T15 Devereaux (4) O'Driscol (8) Counihan (17)
T16 O'Donovan (4) O'Sullivan (6) Counihan (17)
T17 O'Grady (12) Hogan t/k (4) Counihan (17)
T18 O'Grady (12) Murphy (2) Counihan (17)
T19 O'Grady (12) Morrisey (2) O'Callaghan (2)
T20 O'Grady (12) Walt (2) Wakefield (2)
T21 Buckley (7) Leonard (2) O'Dwyer (2)
T22 Buckley (7) Constine (2)
T23 Buckley (7) Caulwell (2)
WT Rogan infant+ (2) O'Keefe+ (6) Concoran+ (6)


4 5 6 7 8 9 10 11
T1 Barrett (6) Hartnett (4) Dromey (4)
T2 Daly (4) Noreen (3)
T3 Fennell (2) Howley t/k (2) Dennehy (2)
T4 Reid (2) Daunt (2)
T5 Cummins (2) Beale (2) Kingston (2)
T6 Hillary (5) Clarke (3)
T7 Keegan (2) Sheehan (2)
T8 Keating t/k (2) Mc Namee t/k (2) Meane (2)
T9 Crowley (3) Riordan (4) Caulfield (2)
T10 O'Keefe (3) Murphy (3)
T11 Keane+ (2) Jeffreys (4) O'Donovan (4)
T12 Sullivan (2) Jennings (2) Moynihan (2)
T14 Good (4) O'Driscol (8) Counihan (17)
T15 Devereaux (4) O'Driscol (8) Counihan (17)
T16 O'Donovan (4) O'Sullivan (6) Counihan (17)
T17 O'Grady (12) Hogan t/k (4) Counihan (17)
T18 O'Grady (12) Murphy (2) Counihan (17)
T19 O'Grady (12) Morrisey (2) O'Callaghan (2)
T20 O'Grady (12) Walt (2) Wakefield (2)
T21 Buckley (7) Leonard (2) O'Dwyer (2)
T22 Buckley (7) Constine (2)
T23 Buckley (7) Caulwell (2)
WT Rogan infant+ (2) O'Keefe+ (6) Concoran+ (6)

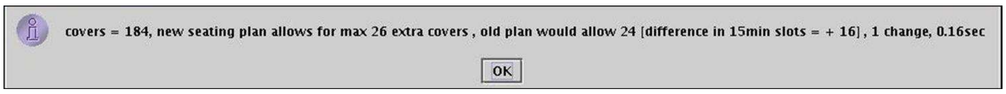
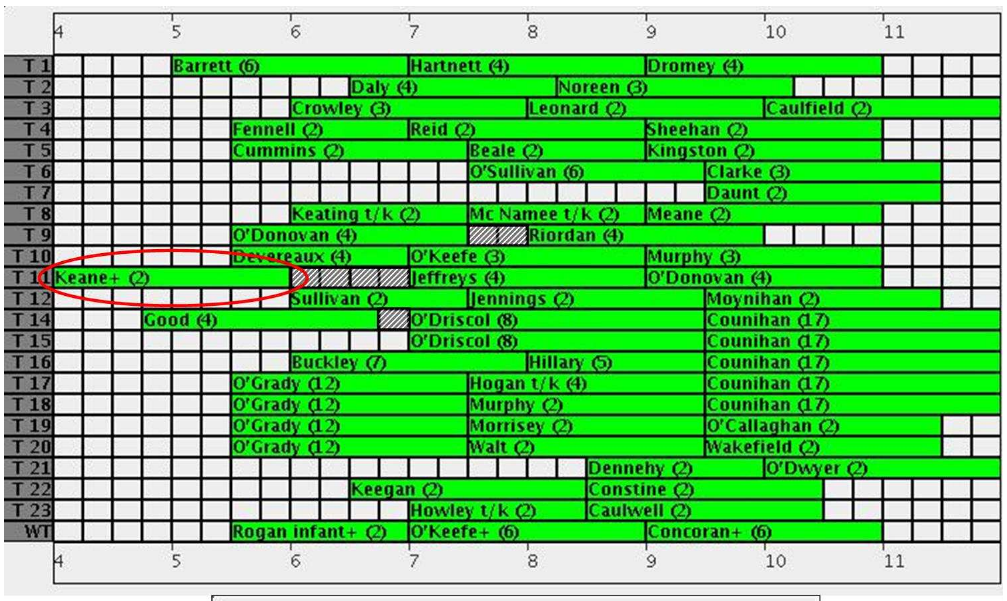
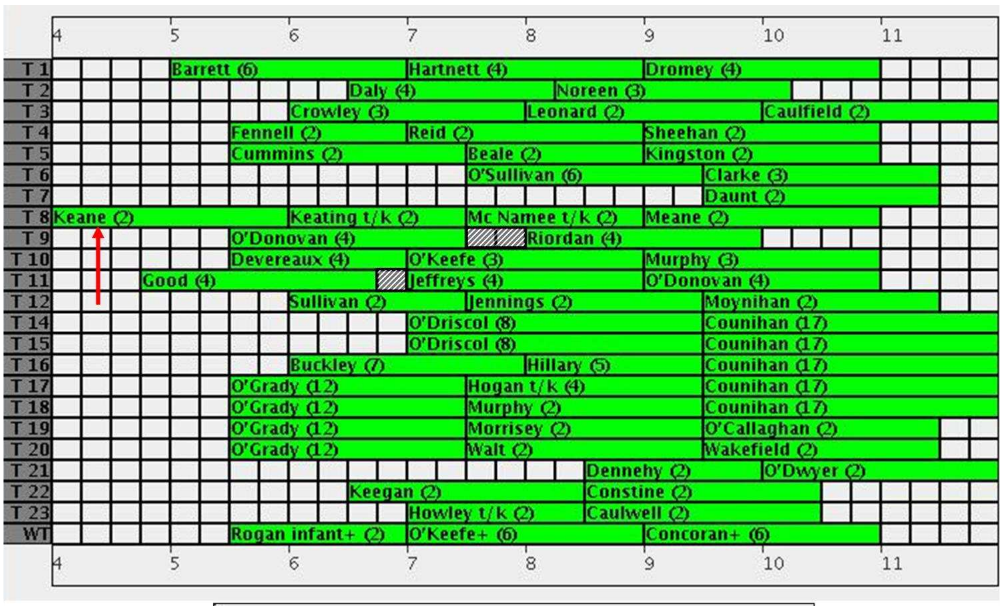

covers = 184, new seating plan allows for max 26 extra covers , old plan would allow 24 [difference in 15min slots = + 16] , 1 change, 0.16sec

Figure 7.8: First improvement: seating plan showing dead zones and poor table usage (top); party *Crowley* moved into a more suitable table (bottom).


4 5 6 7 8 9 10 11
T 1 Barrett (6) Hartnett (4) Dromey (4)
T 2 Daly (4) Noreen (3)
T 3 Crowley (3) Leonard (2) Caulfield (2)
T 4 Fennell (2) Reid (2) Sheehan (2)
T 5 Cummins (2) Beale (2) Kingston (2)
T 6 O'Sullivan (6) Clarke (3)
T 7 Daunt (2)
T 8 Keating t/k (2) Mc Namee t/k (2) Meane (2)
T 9 O'Donovan (4) Riordan (4)
T 10 Devereaux (4) O'Keefe (3) Murphy (3)
T 11 Keane+ (2) Jeffreys (4) O'Donovan (4)
T 12 Sullivan (2) Jennings (2) Moynihan (2)
T 14 Good (4) O'Driscol (8) Counihan (17)
T 15 O'Driscol (8) Counihan (17)
T 16 Buckley (7) Hillary (5) Counihan (17)
T 17 O'Grady (12) Hogan t/k (4) Counihan (17)
T 18 O'Grady (12) Murphy (2) Counihan (17)
T 19 O'Grady (12) Morrisey (2) O'Callaghan (2)
T 20 O'Grady (12) Walt (2) Wakefield (2)
T 21 Dennehy (2) O'Dwyer (2)
T 22 Keegan (2) Constine (2)
T 23 Howley t/k (2) Caulwell (2)
WT Rogan infant+ (2) O'Keefe+ (6) Concoran+ (6)

covers = 184, current seating plan allows for max 32 extra covers


4 5 6 7 8 9 10 11
T 1 Barrett (6) Hartnett (4) Dromey (4)
T 2 Daly (4) Noreen (3)
T 3 Crowley (3) Leonard (2) Caulfield (2)
T 4 Fennell (2) Reid (2) Sheehan (2)
T 5 Cummins (2) Beale (2) Kingston (2)
T 6 O'Sullivan (6) Clarke (3)
T 7 Daunt (2)
T 8 Keane (2) Keating t/k (2) Mc Namee t/k (2) Meane (2)
T 9 O'Donovan (4) Riordan (4)
T 10 Devereaux (4) O'Keefe (3) Murphy (3)
T 11 Good (4) Jeffreys (4) O'Donovan (4)
T 12 Sullivan (2) Jennings (2) Moynihan (2)
T 14 O'Driscol (8) Counihan (17)
T 15 O'Driscol (8) Counihan (17)
T 16 Buckley (7) Hillary (5) Counihan (17)
T 17 O'Grady (12) Hogan t/k (4) Counihan (17)
T 18 O'Grady (12) Murphy (2) Counihan (17)
T 19 O'Grady (12) Morrisey (2) O'Callaghan (2)
T 20 O'Grady (12) Walt (2) Wakefield (2)
T 21 Dennehy (2) O'Dwyer (2)
T 22 Keegan (2) Constine (2)
T 23 Howley t/k (2) Caulwell (2)
WT Rogan infant+ (2) O'Keefe+ (6) Concoran+ (6)

covers = 184, current seating plan allows for max 36 extra covers

Figure 7.9: Improvement after several steps, with party *Keane* fixed (top), unfixed (bottom).

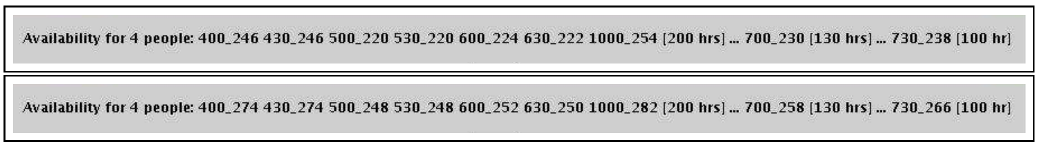
dinner slot. In fact, in the second case the procedure was executed over a more flexible plan - the plan in Figure 7.9 (bottom) was achieved after unlocking party *Keane* from table 11, and after performing a sequence of improvements.

The flexibility associated to dinner slots is important information, as the user can try to sell slots which can preserve more flexible seating plans. For example, in the messages of Figure 7.11, 4:30 appears better than 5:00, 6:00 is more flexible than 5:30 or 6:30, and 10 o'clock is always the most flexible time - in fact, the kitchen closes shortly after 10:00, so a dinner at 10 o'clock does not interact with the flexibility to accommodate future dinners.

```
$CSP(X, D, C) : X = \{P_1, P_2, .., P_n\} \cup P_{n+1},$
                $P_{n+1} = \langle size_{n+1},\ start_{n+1},\ duration_{n+1} \rangle$

$size_{n+1} = new\ request's\ size;$
for $start_{n+1} = 4:00\,p.m.\ ...\ 10:00\,p.m.$ (step $30'$)
    for $duration_{n+1} = stdDuration\ ...\ stdDuration - 1hour$ (step $30'$)
        if    $Solve(CSP,\ timeout_1) ==$ true
              $AVAILABILITY\ [size_{n+1}]\ [start_{n+1}]\ [duration_{n+1}] = 1;$
              $Solution.update();$
              $FLEX\ [size_{n+1}]\ [start_{n+1}]\ [duration_{n+1}] = Solution.flex;$
              while $Improve(CSP,\ timeout_2,\ Solution) ==$ true
                    $Solution.update();$
                    $FLEX\ [size_{n+1}]\ [start_{n+1}]\ [duration_{n+1}] = Solution.flex;$
              break;
```

Figure 7.10: Procedure for computing flexibility by available start time and dinner duration, for a booking request of a given size.



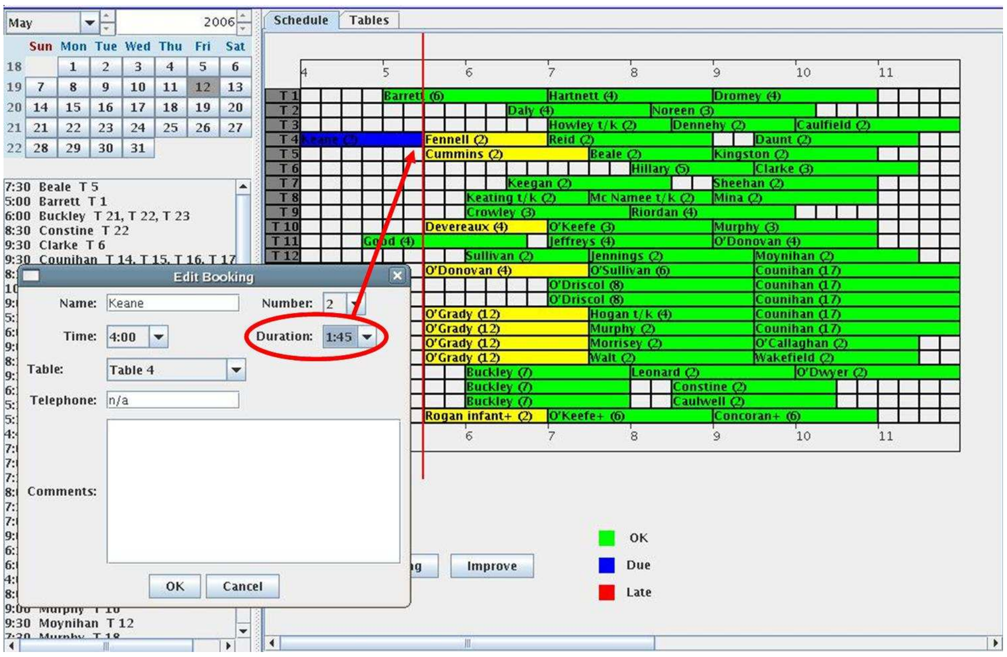
```
Availability for 4 people: 400_246 430_246 500_220 530_220 600_224 630_222 1000_254 [200 hrs] ... 700_230 [130 hrs] ... 730_238 [100 hr]
```

```
Availability for 4 people: 400_274 430_274 500_248 530_248 600_252 630_250 1000_282 [200 hrs] ... 700_258 [130 hrs] ... 730_266 [100 hr]
```

Figure 7.11: Flexibility for the possible time slots available for a dinner for 4 people for Figure 7.8 (top) and for Figure 7.9 (bottom).

7.2.7 Floor management

Figure 7.12 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 1:30hrs to 1:45hrs, and ask the system to search for a reallocation that avoids the conflict.



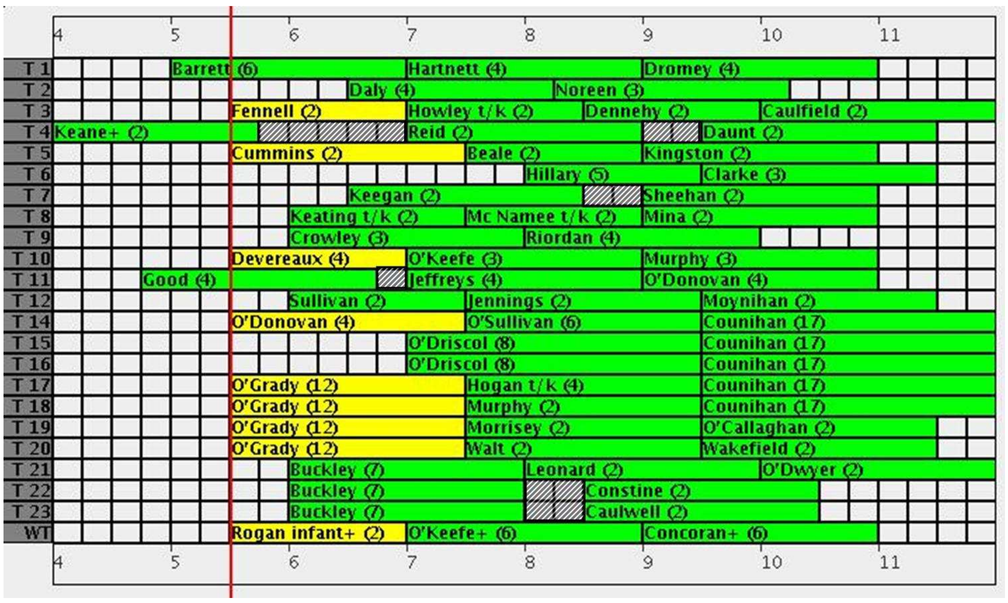
7:30 Beale T 5
5:00 Barrett T 1
6:00 Buckley T 21, T 22, T 23
8:30 Constine T 22
9:30 Clarke T 6
9:30 Counihan T 14, T 15, T 16, T 17
9:30 Moynihan T 12

Name: Keane
Number: 2
Time: 4:00
Duration: 1:45
Table: Table 4
Telephone: n/a
Comments:

Figure 7.12: An instant during floor management, with a late finish (*Keane*, $T4$).

7.2.8 Reaction to changes, stability and optimization

Figure 7.13 (top) represents a first reallocation, while on the bottom we see a seating plan after four improvement iterations. The first reallocation was performed in less than 0.1 seconds and required only one change (party *Fennell* has been moved to table 3). However, the new plan looks poor, as the operation introduced 5 unusable squares in table 4. In Figure 7.13 (bottom) we can again observe the benefit of the improvement, with fewer unusable zones, and more possibilities to seat extra parties.



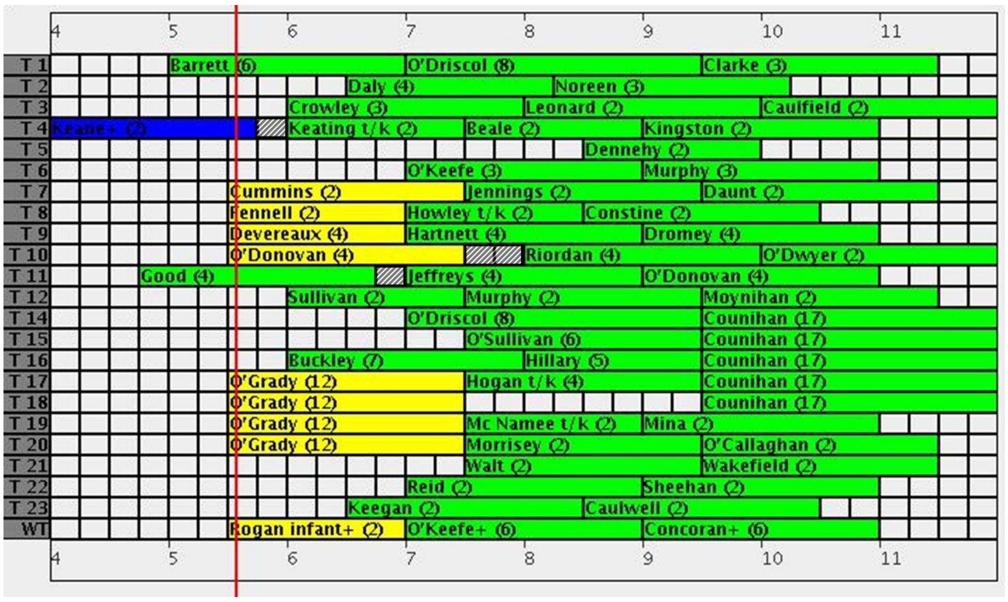


Figure 7.13: Reallocation after a late finish (top); improvement after four iterations (bottom).

Table 7.1 shows the number of changes, the flexibility improvement, and the run time accumulated over the four iterations. Specifically, the iterations have improved the flexibility estimates by steps of 4, 5, 4, and 26, for a total of 39, or $\sim$ 2.5 (2 hour $\times$ 2 people) dinners. In particular, note how the number of unusable squares (dead zones) passes from 14 to just 4. 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.

Table 7.1: Performance over 4 improvement steps for the example of Figure 7.13.

iteration	1	2	3	4
changes	2	3	6	42
improvement [flex$_{WUS}$]	+4	+9	+13	+39
r-time [sec]	1.1	1.3	2.8	10.6

By default, the timeout for each improvement step is set to 10 seconds, partitioned in 7 seconds for search with limited discrepancy (or local) and 3 seconds for unlimited (or global) search. These limits are configurable by the user (e.g. floor manager), who can then trade off between flexibility and stability. For our example, we can observe how the first three iterations are based on pure local search - the run-times are below 3 seconds and there is a contained number of changes. This allows to maintain stability with respect to the original seating plan. Instead, the final step involves a global search (the iteration lasted 7.8 seconds). The improvement is consistent, though it requires many disruptions from the original plan.

7.3 Trial and evaluation

The research prototype software discussed above was in trial in the restaurant Eco from March to September 2006. The main aim of the trial was to test our research in the field, to determine whether our constraint-based solution can support a practical restaurant management tool. Over the trial period, the application had been used by different users from the restaurant staff, such as bookers and floor managers, either with junior or senior experience. The software was utilized in parallel to the standard allocation system (i.e. online, but shadowing the usual allocation process).[†]

7.3.1 Questionnaire overview

At the end of the trial period, E. Fleming (general manager) completed a questionnaire, which we report in Appendix A. In the next part we go through the main points covered in the questions. The questionnaire was designed to get *YES* or *NO* answers, in order to get clear what points have been achieved.

Representing restaurant and table allocations

The first 4 questions were to determine how constraint programming can be used to represent restaurant and table allocations. The manager confirms that the software models the restaurant adequately - e.g. the original tables that can be joined together, and any restrictions that stop tables from being used in a particular way (for example, cannot seat 5 people on table 2 at the same time as 6 people on table 15). Further, the software allows the representation of table allocations adequately - e.g. it describes parties and time slots in a useful way.

Efficiency and quality of the first solutions

Questions 5 to 8 focused on the efficiency and quality of the solutions returned by our constraint satisfaction model (i.e. considering only the first solution returned after each change, and without performing optimization). The manager confirms that the software provides acceptable allocations of parties to tables - i.e. ignoring any issue of flexibility, future arrivals or seating plan preferences, the suggested seating plans are always possible. Further, seating plans are provided in reasonable time, confirming our experimental results on the efficiency and robustness of our solutions presented in Chapter 4.

[†]Our solver was in fact licensed only for research purposes, and not for business use.

Using table configurations

Questions 9 to 11 regarded the representation and use of table configurations. In particular, the manager states that the software joins table correctly (e.g. if there is a large party, or if there are too many parties of size 6 to seat on 6-seater tables). Further, the software correctly manages multiple joining and separation of tables in the same evening (e.g. two parties of two, followed by a party of 4, followed by two parties of two, all on the same 2-seater tables).

Efficiency and quality of the optimized solutions

Questions 12 to 19 are to assess our constraint optimization model based on measuring flexibility. For instance, the manager appreciates how the software proposes flexible seating plans (e.g. how it avoids placing parties on tables that unnecessarily limit the options to seat new arrivals), and how the response happens in good time. Note how this supports our experimental results discussed on Chapter 5 and 6, where we tested the efficiency and quality of our allocation methods. In particular, the manager says that the software provides advice which allows him to maintain a flexible seating plan, it also gives useful advice on how flexible is the current assignment, or on which table to place a booking.

Booking feasibility and negotiation

Questions 20 to 22 concern the efficiency of our software to find whether a booking request is feasible or not, and the support it provides for negotiating alternative booking times in case the original request is infeasible. The manager replied that the response for assessing feasibility is quick, and also that the software recommends sensible alternative times for a booking.

Using future knowledge

Questions 23 to 25 focused on evaluating our attempt to model future knowledge, i.e. to use different booking patterns for counting flexibility on different days of the week. The manager observed how the software makes use of information about how likely are different patterns of bookings, e.g. Saturday evening and

Sunday evening have different patterns, and the system gives an appropriate advice for each one.

Table 7.2 (discussed in Chapter 5) represents booking patterns in the Eco restaurant for Sundays and for Saturdays (or any other days but Sundays). Specifically, the table displays weights (scaled in the range 0 to 4) which represent the distribution of booking requests over the evening session. Note that, for example, the peak of table load (or demand) on Sundays is between 4 and 7 o'clock, while on another day it ranges between 6 and 9 o'clock.

By default the system considers all times from 4 p.m. to 10 p.m. equally likely (i.e. we give a weight of value 1 to each possible time). The user can enable/disable the real booking patterns displayed in Table 7.2 through a submenu of the GUI. Figure 7.14 represents a simple example with real patterns enabled, where the system suggests different seating plans depending on the day. Specifically, Figure 7.14 (top) represents a Saturday, so the correspondent plan was obtained by optimizing (using the *Improve* button) with the Saturday pattern . Similarly, Figure 7.14 (bottom) represents a Sunday and the correspondent plan was obtained with the Sunday pattern.

Managing delays, stability, and infeasibility in floor management

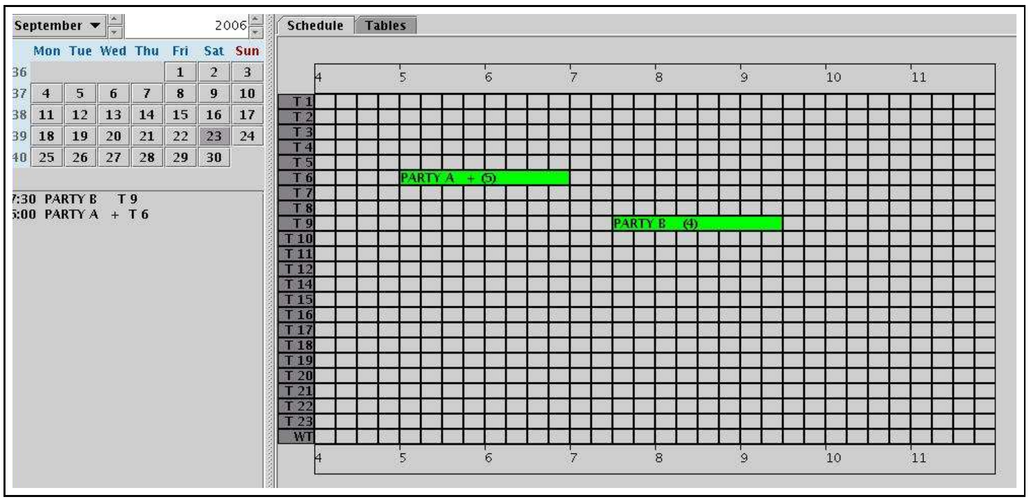
Questions 26 to 31 evaluate the support provided by our software in managing delays (feasible or infeasible), in maintaining stable seating plans when delays or other changes require to disrupt the current plan, and in providing advice when a party (walk-in or booking) cannot be seated.

The manager stated that the software provides useful advice when a seating plan has to be reconfigured. For example, if a party is late in arriving, or if a dinner takes longer than expected, and this causes a conflict with the dinner following on the same table, the system can often find a seating plan reallocation which preserves start times and durations of all the future dinners, so that no party gets delayed.

Further, when a seating plan has to be reconfigured, the software manages to keep most of the plan as it was. This allows to maintain stability, i.e. to avoid frequent table reallocations and, in particular, frequent reconfiguration of the restau-

Table 7.2: Distribution of bookings over time in Eco, for Sundays and Saturdays.

clock	400	430	500	530	600	630	700	730	800	830	900	930	1000
time-unit	0	2	4	6	8	10	12	14	16	18	20	22	24
Sundays	4	4	4	4	4	4	4	3	2	2	1	1	0
Saturdays	1	1	2	2	4	4	4	4	4	4	4	2	1


7:30 PARTY B T 9
5:00 PARTY A + T 6

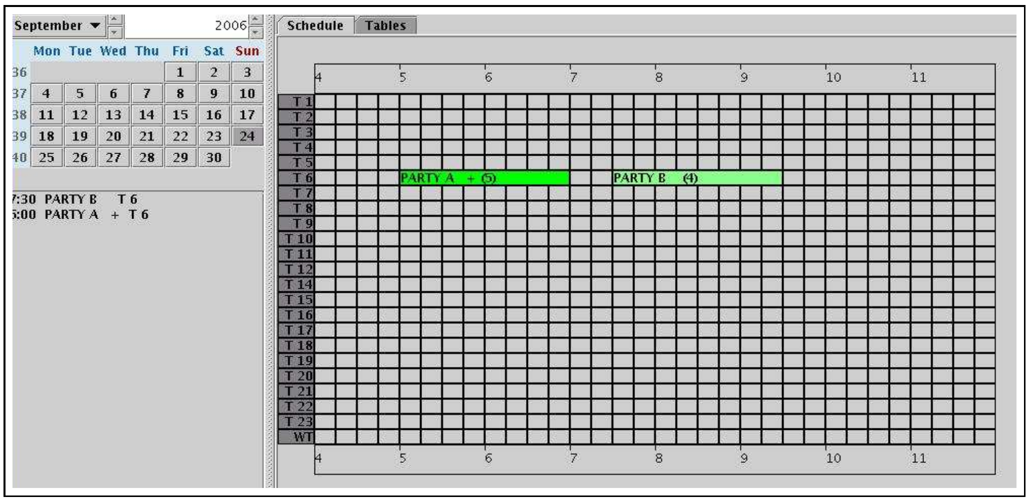

7:30 PARTY B T 6
5:00 PARTY A + T 6

Figure 7.14: Allocation optimized for a Saturday (top); for a Sunday (bottom).

rant layout, which can be quite annoying for both customer and staff.

The manager also stated that the software does not help to seat in the cases when the software finds that there is no solution, e.g. when a party cannot be seated without delaying or cancelling other bookings. In the real restaurant, instead, they would delay people and fit everybody in. This is the only negative answer to the questionnaire, and was expected, since our research did not develop any policy on managing delays - which will be part of our future research.

Finally, the manager said that the software gives useful advice when a party (walk-in or booking) cannot be seated. It does this through the availability function, which provides the list of alternative times still available.

Graphical user interface

The final questions 32 to 34 focused on the graphical interface. In the questionnaire we did not concentrate on the information advice given by the program interface, though we have included these questions about the interface at the end. The current GUI has been developed at a basic level, with the only scope to allow the evaluation of our research. As expected, the main comment on the GUI was that it needs to be improved, i.e. more user friendly.

7.4 Chapter summary

In this chapter we presented the application software prototype which had been in trial in the restaurant Eco for six months from March 2006. The solver engine was developed in C++, using Ilog Solver 6.0, and based on the knowledge acquired from the management and staff from Eco. The GUI was specified by the author, and implemented in Java 1.5 by James Lupton (software engineer). The system was running in Linux (Fedora) on a laptop with a 1.6 GHz Pentium processor.

The aim of the trial was to assess our study and research on modelling and solving the restaurant table management problem. At the end of the trial we prepared a questionnaire for the restaurant, which was then completed by E. Flemming (general manager of Eco). A copy is reported in Appendix A.

The replies we obtained let us conclude that our system represents a practical solution to support and improve table management for real restaurants. In particular, inexperienced users can now take bookings, we turn down fewer requests in booking, we maximize table usage, we support the booker with availability data, which can be used to guide negotiation, and we support the reaction to unplanned events, maintaining feasible, flexible, and stable seating plans.

Chapter 8

Future work

8.1 Search algorithm

In this thesis, we have modelled table management as a constraint satisfaction problem. In particular, we have developed a search approach based on multiple heuristics and time slicing (MH, Chapter 4), which we have demonstrated improves search efficiency and robustness for solving the table management problem compared to other recommended approaches. The different ordering heuristics and the different time limits were not tuned, but were generated by inspection of the problem characteristics, so better performance should be achievable. Future work should then focus on optimizing the set of heuristics, the order heuristics are selected within the same set, and the sequence of time limits. Further, in the current version, each restart resets the search process to the root of the search tree. More improvement may be obtained by extending the model to cache the search state of each ordering heuristic after each round of restarts, so that the heuristics can then resume it on the successive round. However, caching requires extra memory for storing the partial path achieved by each heuristic, and would also introduce an overhead in run time for maintaining each state. This may become too expensive for large problems. Future research should investigate whether, when, and to what extent, savings in search outweigh the overhead of caching. Finally, we also intend to investigate whether MH does perform better on insoluble problems (as indicated by the scheduling results in Chapter 4).

8.2 Table mix

Our constraint model supports table configurations, i.e. it models tables that can be joined together to serve larger parties. The model has been applied to the restaurant Eco, which has a specific table mix of 23 tables, each with a certain capacity and combinability. Although Eco is satisfied by the flexibility of the current table mix, a different mix of table sizes and combinability may increase the number of people the restaurant can accommodate. Future research should focus on table mix optimization, e.g. using Thompson's simulator [90]. Thompson's simulator generates optimal table mixes for restaurants with single (non-combinable) tables. We should extend the simulator to optimize table mix by both table sizes and combinability. We should then evaluate the results on the Eco restaurant, comparing the performance achieved using the current table mix (i.e. the results presented in this dissertation) to the performance achieved using an optimal table mix.

8.3 Flexibility

In Chapter 5, we presented our constraint model for seating plan optimization which is based on measuring (and maximizing) seating plan flexibility. We designed three different measures, based on usable-start-times, dead-zones, and on potential number-of-seatings. Better performance might be achievable using some combination of these three measures, which have been tested only singularly so far. For example, considering the first measure, there may be a pair of feasible seating plans, both measuring the same value of usable-start-times, but with different dead-zone values. The seating plan with lower dead-zone value should then be selected, but the current measure cannot distinguish between the two plans. A first extension could be to maximize usable-start-times, breaking ties by minimum dead-zone (and/or by maximum number-of-seatings). Otherwise, the overall objective could be to maximize a linear combination of the three measures.

Flexibility has been weighted by the expectation of future table demand over time, where weights are based on booking patterns retrieved from past booking sheets. However, booking sheet data represents the history of the final table allocations. There is a clear need for better estimates of table demand - e.g. recording a period of table requests, regardless of whether the (potential) customers are actually accepted or rejected.

Further, our current measures weight a slot in a table Tb at a generic time t by the table capacity and by the expected number of bookings (of any size) arriving at time t. Future work will consider flexibility measures which weight the same table slot by the expected number of bookings arriving at time t and of size suitable for the table. For example, if on a Sunday there is a dinner slot available from 7 p.m. on a two seater table then our current measures weights that slot by the expected number of bookings of any size at 7 p.m. on a Sunday. In a future version, for the same dinner slot the weight should consider the expected number of bookings of size 2 at 7 p.m. (again, based on the Sunday's pattern).

Finally, among the optimization methods from the literature, we should investigate how *sampling* (Chapter 2) can be applied to the restaurant problem. A first attempt would be to decide which table to allocate to the next party waiting in the queue based on consensus [10]. For example, consider a party P_i with a reservation for 7 o'clock. When P_i arrives and has to be seated, the current seating plan comprises parties that arrived and got seated earlier, i.e. whose table allocation is then fixed, and parties that have a reservation for later times, i.e. whose table allocation is only provisional. Some unknown table requests (walk-ins or new bookings) are also expected to arrive later in the dining session. The aim is to decide the table where to seat P_i (now) so that each party in the current seating plan is guaranteed a table, and the maximum number of future table requests can be satisfied. In order to do that, we can resolve the allocation of the list of parties in the current seating plan extended with different samples of future arrivals, and then select the table where P_i appears most over the seating plans that solved the extended problems.

8.4 Managing critical changes

In this thesis, we defined the restaurant problem assuming that dinners are assigned fixed start and end times, i.e. time slots of specific durations that cannot be delayed. Future work concerns improving the solving algorithms for the cases where a (critical) change allows no feasible reallocation. For example, a change like a dinner lasting longer than expected can cause the current allocation to become infeasible, i.e. the booking time and expected dinner duration of all the other parties cannot be preserved. In this case, the current version simply replies that there is no possible reallocation, while, in the real case, the floor manager must fit everybody in and make future diners wait.

Our current model can be used to notify the user when such types of changes are infeasible (i.e. critical), but then the user has to manually insert the delays into the model in order to resume a feasible seating plan. The system could be made more helpful, supporting the automatic management of critical changes. For example, a future version could model what are the standard policies to manage delays (e.g. seat first the party that has been waiting most), and further investigation may aim to improve the current approaches (e.g. provide allocations that minimize the delays for future parties).

8.5 Supporting robustness

The frequency of critical changes can be reduced by increasing seating plan robustness. A first approach could be based on the addition of temporal protections (or slack times) to dinner slots [30] (Chapter 2), so that critical changes (especially delays) are more likely to be absorbed without delaying future parties. Another approach could be computing some form of (a, b)-super solutions [55] (also Chapter 2). For example, a first attempt could be to look at a as the number of parties simultaneously changing either size, start time, or duration, and at b as the maximum number of other parties requiring a delay in start time in order to accommodate any a changes. Changes generally happen one by one, so we can set $a = 1$. The goal, in terms of robustness, could be to maintain seating plans that maximize the number of single changes repairable with zero delays in start time (i.e. with no increase in waiting time). Asking for high robustness may degrade the flexibility of seating plans, reducing the potential number of people that can be accommodated. In practice, the overall objective function should then balance robustness with flexibility.

8.6 Supporting stability

When accommodating a new change (e.g. a new booking, or a delay) on the current seating plan, our system allows a trade off between flexibility to satisfy future requests, and stability to control the number of plan disruptions. Specifically (Chapter 7), the search process performs a first phase, looking for solutions similar to the original seating plan, and then moves to a second phase of global search, in case no stable solutions exist.

Instability could be partially prevented using (again) temporal protections [30] to dinners, or (a, b)-super solutions [55] (Chapter 2). For the latter, as for the preceding case on critical changes, we could look at $a = 1$ as the number of parties simultaneously changing either size, start time, or duration. In this case, the value b is the maximum number of other parties requiring a change in table allocation (rather than a delay in start time) in order to accommodate any a changes. The goal, in terms of stability, could be to maintain seating plans that maximize the number of single changes repairable with zero table reallocations. Ultimately, for a practical use, the overall objective should balance stability with both flexibility and robustness, e.g. maximizing the potential number of people that can be accommodated, while limiting delays and table reallocations.

A further step would be to extend the stability model to weight table reallocations by type. For example, reallocations which require moving tables around, can further annoy the customer, and can also require extra effort and time for the set up. Therefore, seating plan disruptions that change the restaurant layout should be penalized more compared to disruptions that maintain the same layout.

Chapter 9

Conclusions

THESIS:

Constraint programming can be used as a tool to support, enhance, and automate uncertain and highly dynamic restaurant table management.

Specifically, constraint programming can be used:

- *to model table management, and careful modelling can improve efficiency;*
- *to model and solve the underlying static decision problem;*
- *to model table (re)configurations and seating plan flexibility;*
- *for complex or stable seating plan reallocations;*
- *to model knowledge on future demand to build more flexible seating plans;*
- *to exploit diner's start time flexibility to preserve seating plan flexibility;*
- *to improve robustness in managing uncertainty (on demand and changes).*

In this dissertation, we presented a solution based on constraint programming for enhancing restaurant table management. The solution was designed using information from a real restaurant (Eco), was tested using computer simulations, and a software prototype was finally evaluated with trials at the restaurant. The contributions we have provided in this dissertation, supported by both simulation and real results, are a strong support to our thesis.

9.1 Summary of contributions

We conclude this chapter with a summary of the main contributions provided, chapter by chapter, in this dissertation.

9.1.1 Representing restaurant table management

Restaurant table management (Chapter 2) is a complex and dynamic problem: restaurants must manage reservations and unexpected events in real-time, making good use of resources, and providing good service to customers. In this dissertation, we represented the dynamic table management problem as a sequence of static problems linked by changes. We modelled the underlying static problem as a subclass of scheduling with fixed start and end times. This subclass of scheduling is representable using μ-coloring on interval graphs, which is an NP-complete problem (Chapter 3).

9.1.2 Modelling and solving the static decision problem

In Chapter 4, we presented the basic constraint model used to solve the underlying static decision problem. The goal was to provide efficient and robust solutions over distributions of static instances.

Careful modelling can improve efficiency - The goal was achieved through the design of specialized constraints, and the development of a search approach based on multiple heuristics and time slicing (MH). For the original problem, dimensioned upon the Eco restaurant, we can find seating plans accommodating full booking sheets of 200 covers in few tenths of a second on average.

9.1.3 Modelling combinable tables and seating plan flexibility

Chapter 5 extended the model to represent table configurations and to perform seating plan optimization. The new model optimizes solutions based on measures of flexibility, advising which tables have to be allocated or joined, and when, in order to get seating plans that are more flexible for accepting future table demand.

Careful modelling can improve efficiency - Through the design of new specific constraints, and of anytime algorithms which can provide sensible solution improvements in acceptable time, we achieved a good balance between optimization and search efficiency, i.e. a reasonable ratio improvement/time.

9.1.4 Managing uncertainty on table demand and changes

In Chapter 6, we tested our models on the dynamic problem, through computer simulations of booking and floor management sessions.

Maintaining stable seating plans - We first extended our search algorithm to consider solution stability. The algorithm performs a first phase constraining the search to a few changes, i.e. exploring the neighborhood of the previous solution, and switching to search with unlimited changes only if necessary.

Providing complex reallocations to improve turnover and reduce delays - An important contribution to the ability to solve the dynamic problem concerns the fact that our model can provide complex seating plan reallocations, requiring many changes, which allows more options to accommodate future table requests (bookings or walk-ins), and more options to accommodate changes (e.g. delays) without delaying other parties.

Providing flexible seating plans to improve turnover - Another important contribution to the ability to solve the dynamic problem concerns our flexibility based optimization, i.e. the fact that the solutions we provide are designed to be flexible for the accommodation of future table requests. When we performed optimization online - i.e. each time a new table request was allocated - results on both floor and booking simulations showed a further improvement in the final number of people accepted.

Exploiting diner's start time flexibility to improve turnover - We found a potential benefit of using our optimization model to exploit customer's start time flexibility, i.e. when some customers are flexible over the time to consume their dinner. For instance, we assumed some customers were available to accept any among three different booking times (t, and t + or - 1 hour), and when accommodating those customers we selected the time for which the seating plan preserved higher flexibility to accept future booking requests. Doing this, the final number of people allocated was considerably higher, and the reservation target of 180 covers was reached with significantly fewer booking requests, compared to the case with no flexibility over start times.

Accurate measures of flexibility - We evaluated three versions of our optimization model based on three flexibility measures designed in Chapter 5. The three versions are based, in the order, on maximizing usable start times, minimizing dead zones, and maximizing the number of potential seatings. Results showed the three optimization models have similar and good performances, which indicates that the measures are all accurate in representing the real flexibility.

Modelling future knowledge to increase flexibility and turnover - A further improvement was achieved from our first attempt to make use of booking patterns, i.e. from weighting seating plan flexibility by the expected distribution of future table demand. The results were noticeably improved when we assumed some booking requests with flexible start times.

9.1.5 An interactive tool for restaurant table management

In Chapter 7, we described the implemented research prototype software.

Supporting, enhancing, and automating current systems - The software integrates allocation and optimization facilities - the user can interact with the system, controlling table allocation, while receiving advice from the underlying models. The software had been in trial at the restaurant for six months and a questionnaire was completed by the general manager. The questionnaire was intended to validate the results achieved in simulation, i.e. to assess the applicability of our research to the real restaurant industry. The evaluation was positive. Specifically, the replies to the questionnaire report that the software models the restaurant adequately, provides acceptable seating plans in reasonable time, can join and separate tables correctly, proposes flexible seating plans in reasonable time, reports quickly whether or not a booking request can be accepted, recommends sensible alternative times for a booking, and provides useful advice when a seating plan has to be reconfigured.

9.2 Final discussion

Restaurant Table Management (RTM), like many real world problems, is dynamic and uncertain. Several solving techniques have been introduced in the past to tackle different issues in dynamic problem solving (see Background, Chapter 3). The main goals involve: (i) providing quick reactions to changes; (ii) optimizing plans by reasoning on possible future developments of the problem; (iii) maintaining stable solutions; (iv) providing solutions that are robust in accommodating changes at little cost.

In Chapter 3, we presented the state of the art in dynamic problem solving, and we observed how the different approaches tend to focus mainly on one single goal at a time. For example, Probabilistic [36], Stochastic [67], and Branching [38] CSPs, as well as the Sampling methods [10] [11] [12] [26], mainly focus on (ii); Local Changes [91], and the works by Ran et. al. [79] and by Petcu et. al. [74] on optimally stable solutions, focus on (iii); finally, the works on Slack Times [30] and Just In Case scheduling [35], and the Super Solutions framework [55], focus more on (iv). In RTM, however, multiple goals often need to be achieved in a single solution - e.g. table reallocations must be performed in reasonable time, and solutions might require a balance between maintaining seating plan stability and maximizing the flexibility to accommodate future bookings.

Further, many of the techniques described above are purely *proactive*, i.e. their decisions are based on accurate models of the future. But the future is hardly predictable in a restaurant environment. Furthermore, given the high and uncertain dynamism of the problem, maintaining an up to date model of the future could be practically infeasible.

In this dissertation, we proposed a solution that is a balance between a *reactive* and *proactive* approach. The solution guarantees robust real-time reactions to changes, but we can also control stability, and perform time-efficient seating plan optimization in face of uncertainty. Optimization is based on a constraint model for flexible table allocations. Our flexibility model is a cheaper and more heuristic based model, compared to the existing methods, but it represents an efficient and effective solution to maximize resource usage with respect to the future.

In conclusion, with our research we achieved a practical solution for enhancing restaurant table management. The research is valuable for restaurateurs, but also for the constraint programming community. Restaurant table management has no history in the literature of constraint programming. We believe that this dissertation represents a good and already advanced baseline for tackling an interesting and novel, dynamic and highly uncertain problem. The constraint based model proposed in this thesis represents a successful case of research applied to a real-world problem. The model integrates reaction efficiency, optimization, stability, and robustness, which are four major goals concerning dynamic problem solving.

9.3 Conclusions generalized

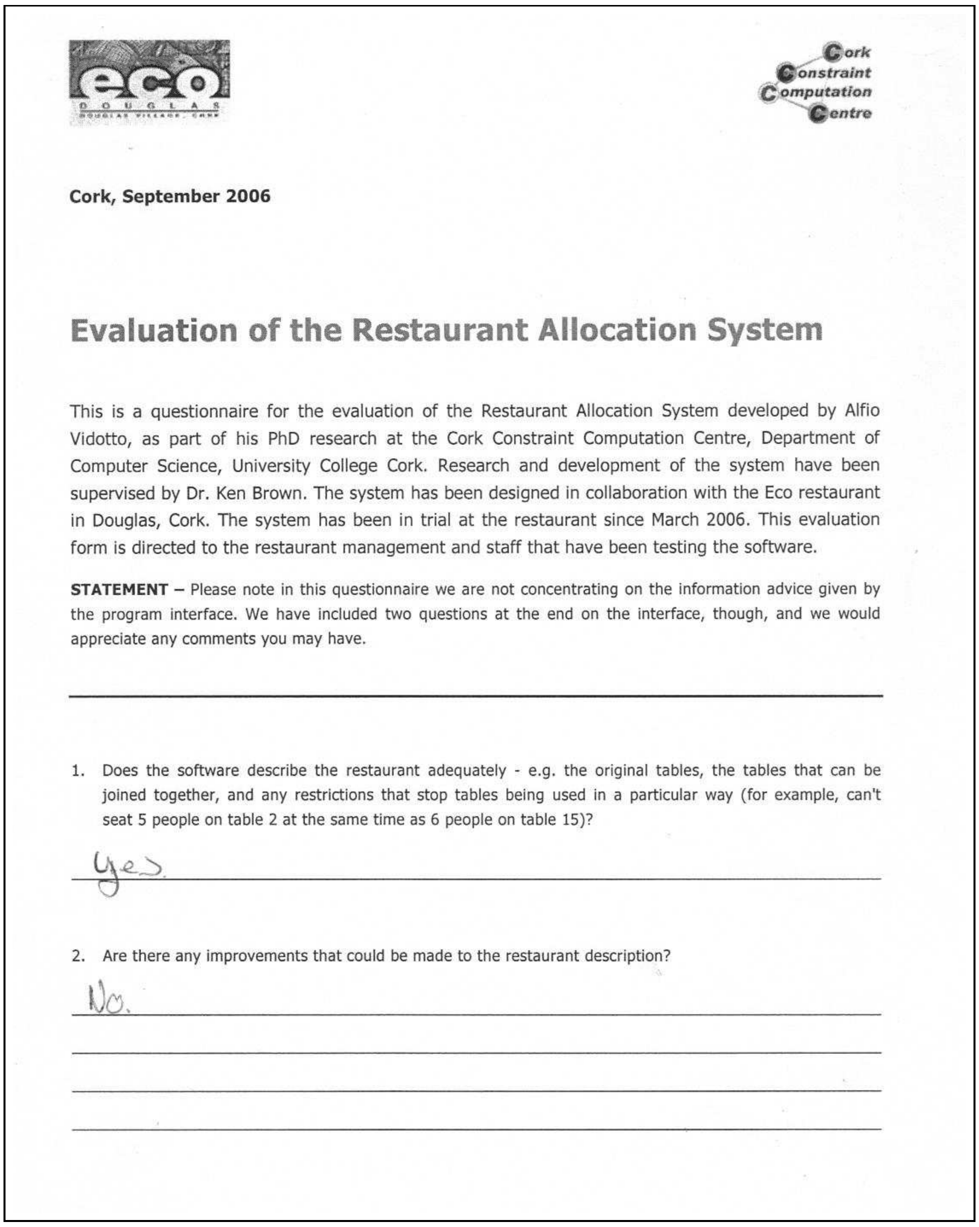
In Restaurant Table Management (RTM), there are limited resources (tables), and we must manage customer reservations and customer usage of those resources. Although the current system is developed for RTM, the underlying techniques are general, and could be applied to many other reservation systems in the service sector. Hotels, car rental agencies, and call centers are only a few categories of service providers that, like restaurants, have to manage the allocation of a variety of customers to multiple reconfigurable resources of different capacity. In all these problems, customer demand and customer usage of resources is uncertain. The service provider does not know how many reservation requests will be made, nor how strictly the customer will stick to the agreed reservation. The service provider aims to maximize the use of its resources in the long term, which involves a trade-off between short term utilization of the resource and long-term customer satisfaction.

We envisage that the research proposed in this dissertation can be used to manage this generalized class of service allocation problems. For example, as in RTM, several service providers are faced with two main problems: (i) how to decide quickly whether a requested service reservation or required change in resource usage is feasible; (ii) how to decide whether a requested service reservation or required change in resource usage is desirable.

Our algorithms provide a fast method of checking the feasibility of changes, and propose a new acceptable allocation whenever a change can be accommodated. This allows the service providers to quickly reconfigure the resource allocation plan when a customer changes the resource usage (e.g. a car rental agency may have a client requesting a car upgrade, or an extension of the rental period), or when a customer requests a new service (e.g. a new rental request for a 7-seater car for a week). Further, our algorithms optimize the response to a requested change, where the response may be to reject the change and suggest alternatives, or to accept the change and provide a new optimized plan.

In conclusion, to our knowledge, the solution proposed in this dissertation is the first which allows automated real-time reconfiguration of reservation plans in the service sector, and which offers optimized reservation plans. In scientific terms, our use of constraint programming algorithms for optimized reconfiguration in the face of uncertainty is also novel.

Appendix A



ECO DOUGLAS

Cork Constraint Computation Centre

Cork, September 2006

Evaluation of the Restaurant Allocation System

This is a questionnaire for the evaluation of the Restaurant Allocation System developed by Alfio Vidotto, as part of his PhD research at the Cork Constraint Computation Centre, Department of Computer Science, University College Cork. Research and development of the system have been supervised by Dr. Ken Brown. The system has been designed in collaboration with the Eco restaurant in Douglas, Cork. The system has been in trial at the restaurant since March 2006. This evaluation form is directed to the restaurant management and staff that have been testing the software.

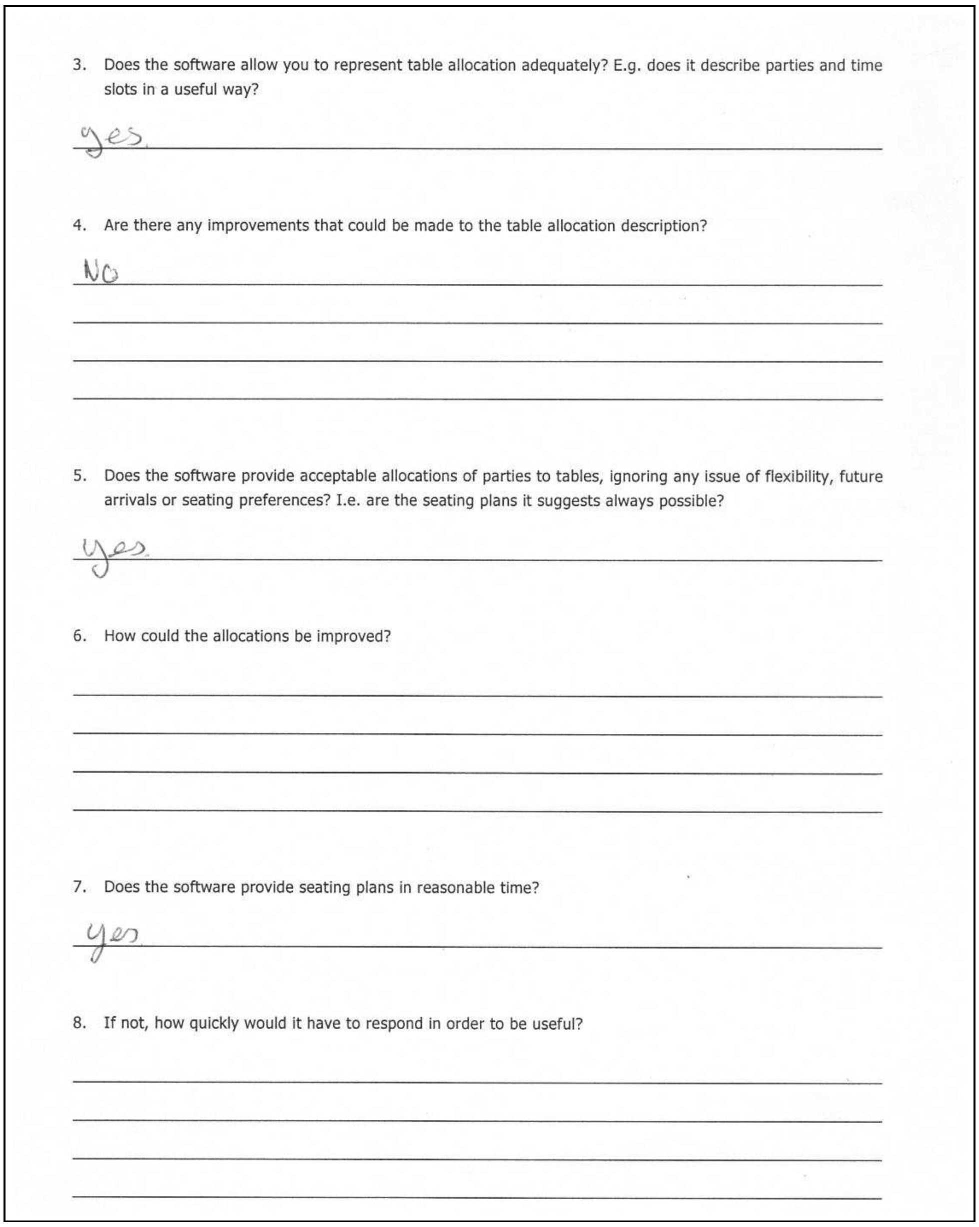
STATEMENT – Please note in this questionnaire we are not concentrating on the information advice given by the program interface. We have included two questions at the end on the interface, though, and we would appreciate any comments you may have.

1. Does the software describe the restaurant adequately - e.g. the original tables, the tables that can be joined together, and any restrictions that stop tables being used in a particular way (for example, can't seat 5 people on table 2 at the same time as 6 people on table 15)?

   yes.

2. Are there any improvements that could be made to the restaurant description?

   No.



3. Does the software allow you to represent table allocation adequately? E.g. does it describe parties and time slots in a useful way?

yes.

4. Are there any improvements that could be made to the table allocation description?

No

5. Does the software provide acceptable allocations of parties to tables, ignoring any issue of flexibility, future arrivals or seating preferences? I.e. are the seating plans it suggests always possible?

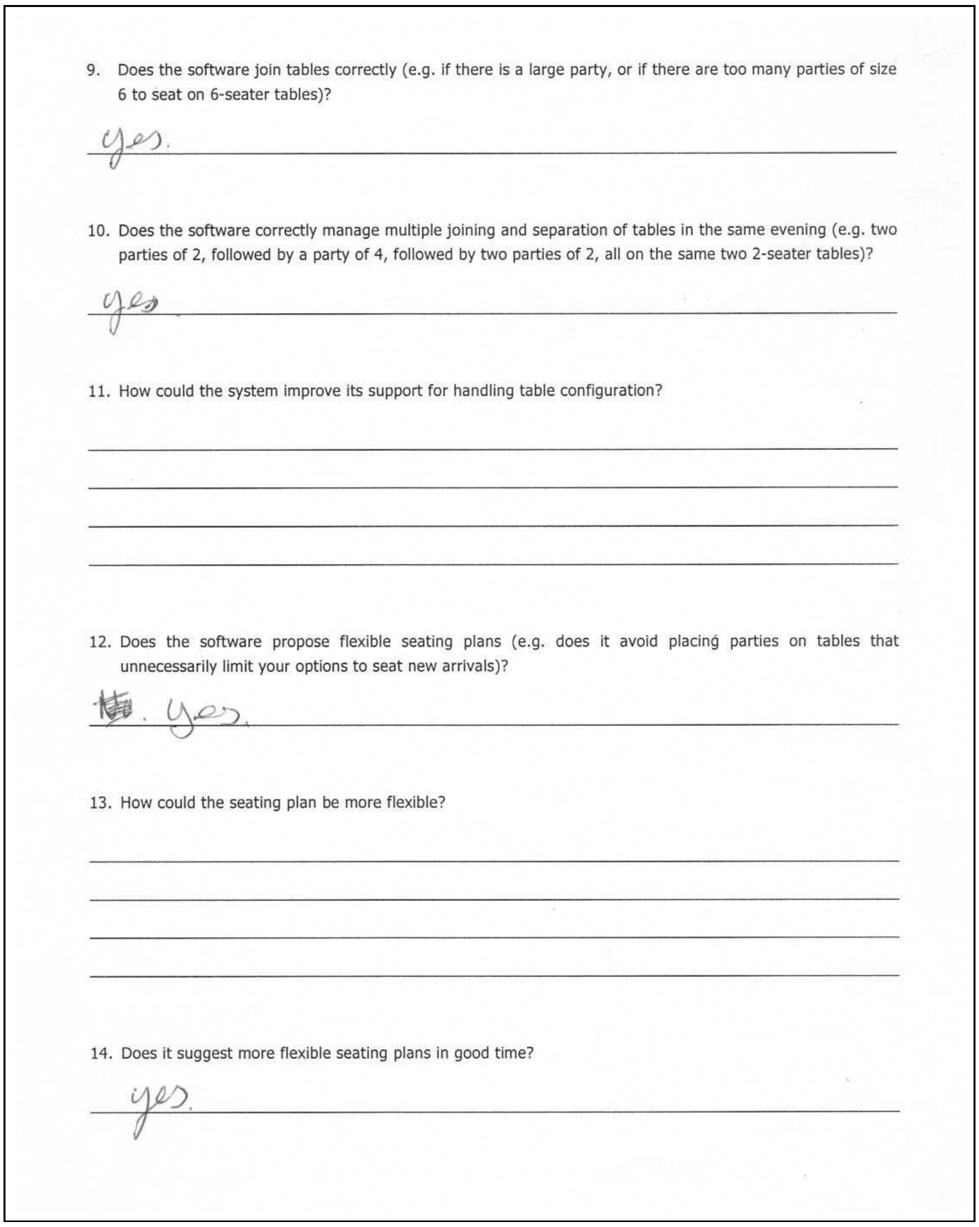
yes.

6. How could the allocations be improved?

7. Does the software provide seating plans in reasonable time?

yes.

8. If not, how quickly would it have to respond in order to be useful?



9. Does the software join tables correctly (e.g. if there is a large party, or if there are too many parties of size 6 to seat on 6-seater tables)?

yes.

10. Does the software correctly manage multiple joining and separation of tables in the same evening (e.g. two parties of 2, followed by a party of 4, followed by two parties of 2, all on the same two 2-seater tables)?

yes

11. How could the system improve its support for handling table configuration?

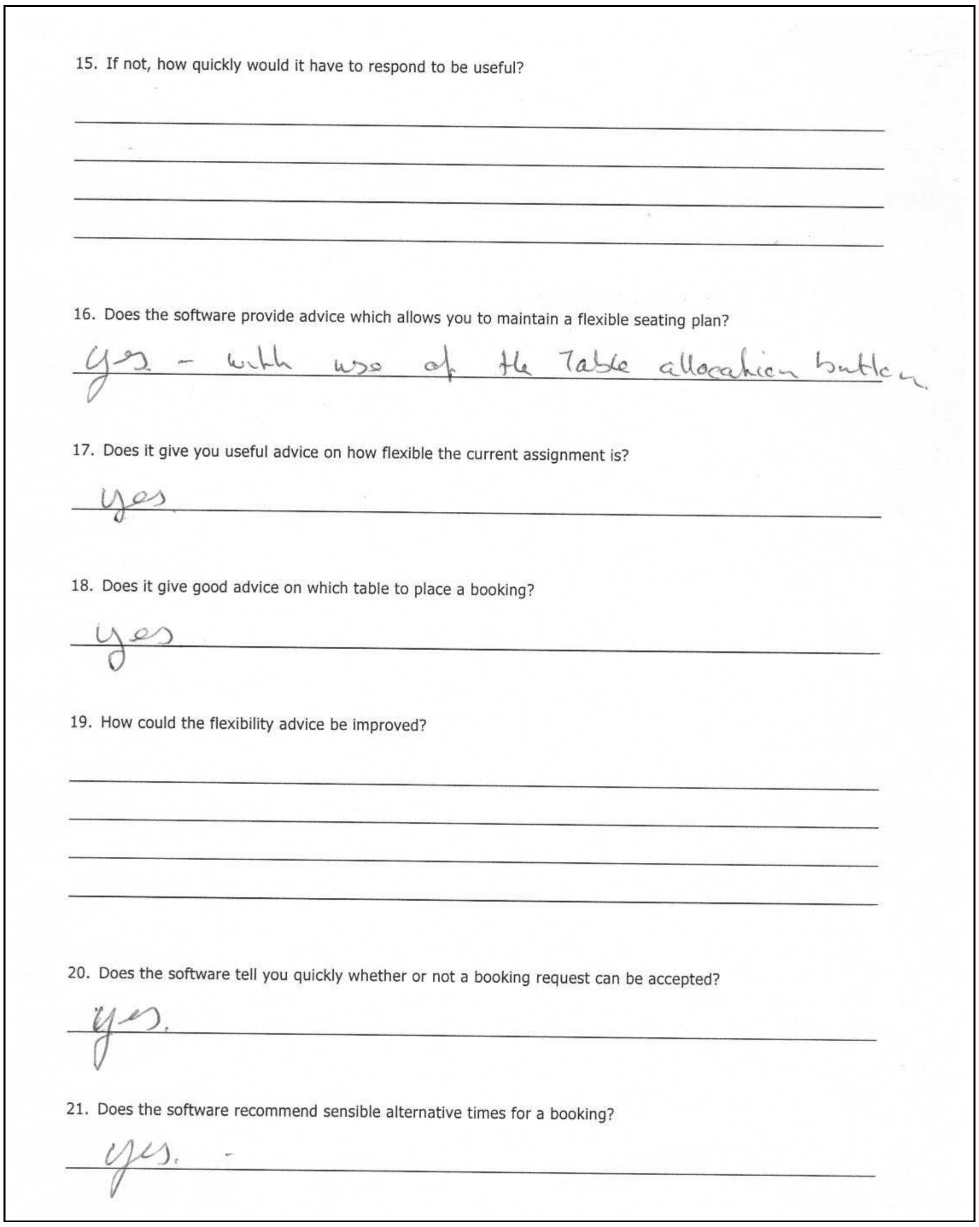
12. Does the software propose flexible seating plans (e.g. does it avoid placing parties on tables that unnecessarily limit your options to seat new arrivals)?

[illegible]. yes.

13. How could the seating plan be more flexible?

14. Does it suggest more flexible seating plans in good time?

yes.



15. If not, how quickly would it have to respond to be useful?

16. Does the software provide advice which allows you to maintain a flexible seating plan?

yes - with use of the Table allocation button.

17. Does it give you useful advice on how flexible the current assignment is?

yes.

18. Does it give good advice on which table to place a booking?

yes

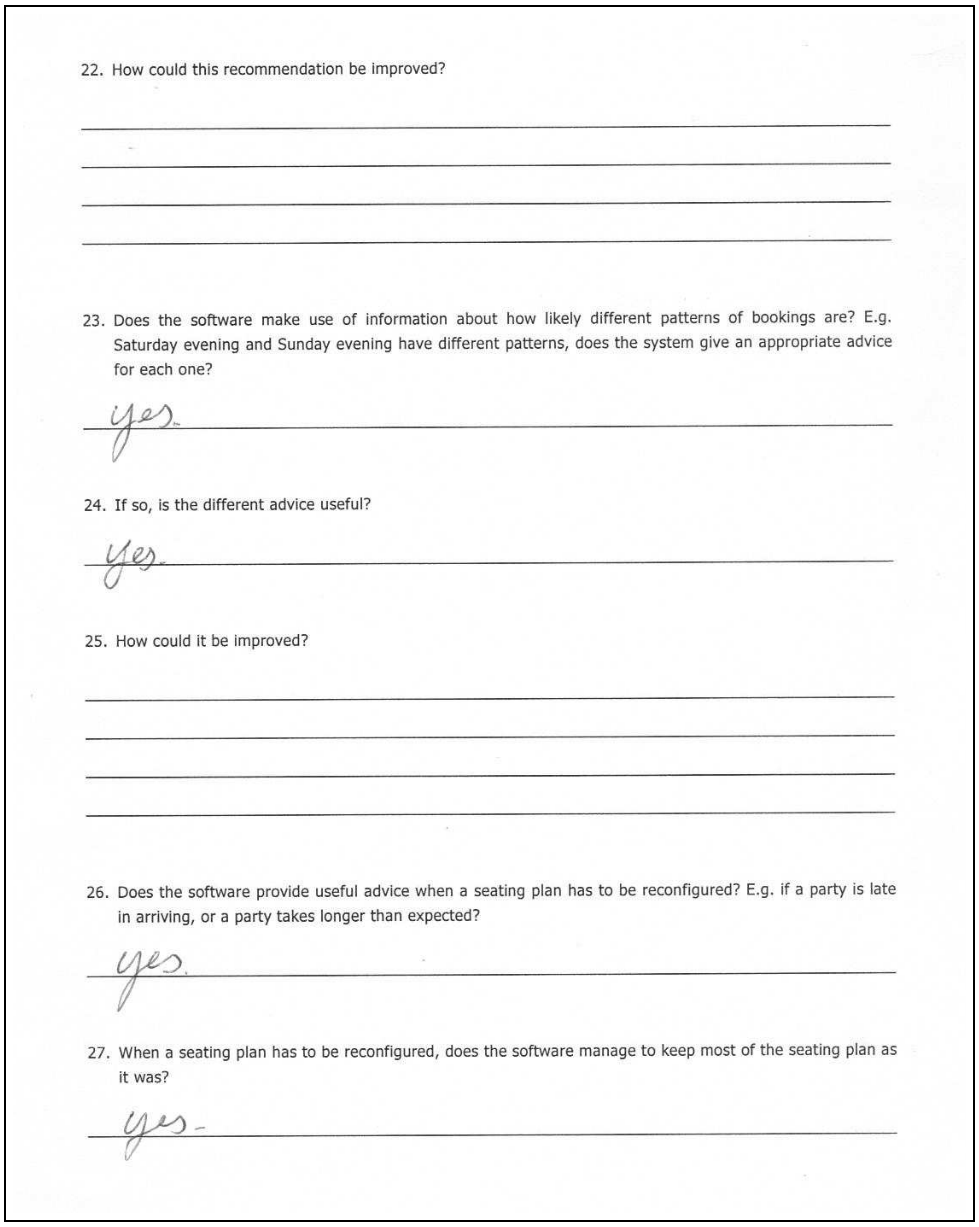
19. How could the flexibility advice be improved?

20. Does the software tell you quickly whether or not a booking request can be accepted?

yes.

21. Does the software recommend sensible alternative times for a booking?

yes. -



22. How could this recommendation be improved?

23. Does the software make use of information about how likely different patterns of bookings are? E.g. Saturday evening and Sunday evening have different patterns, does the system give an appropriate advice for each one?

yes.

24. If so, is the different advice useful?

yes.

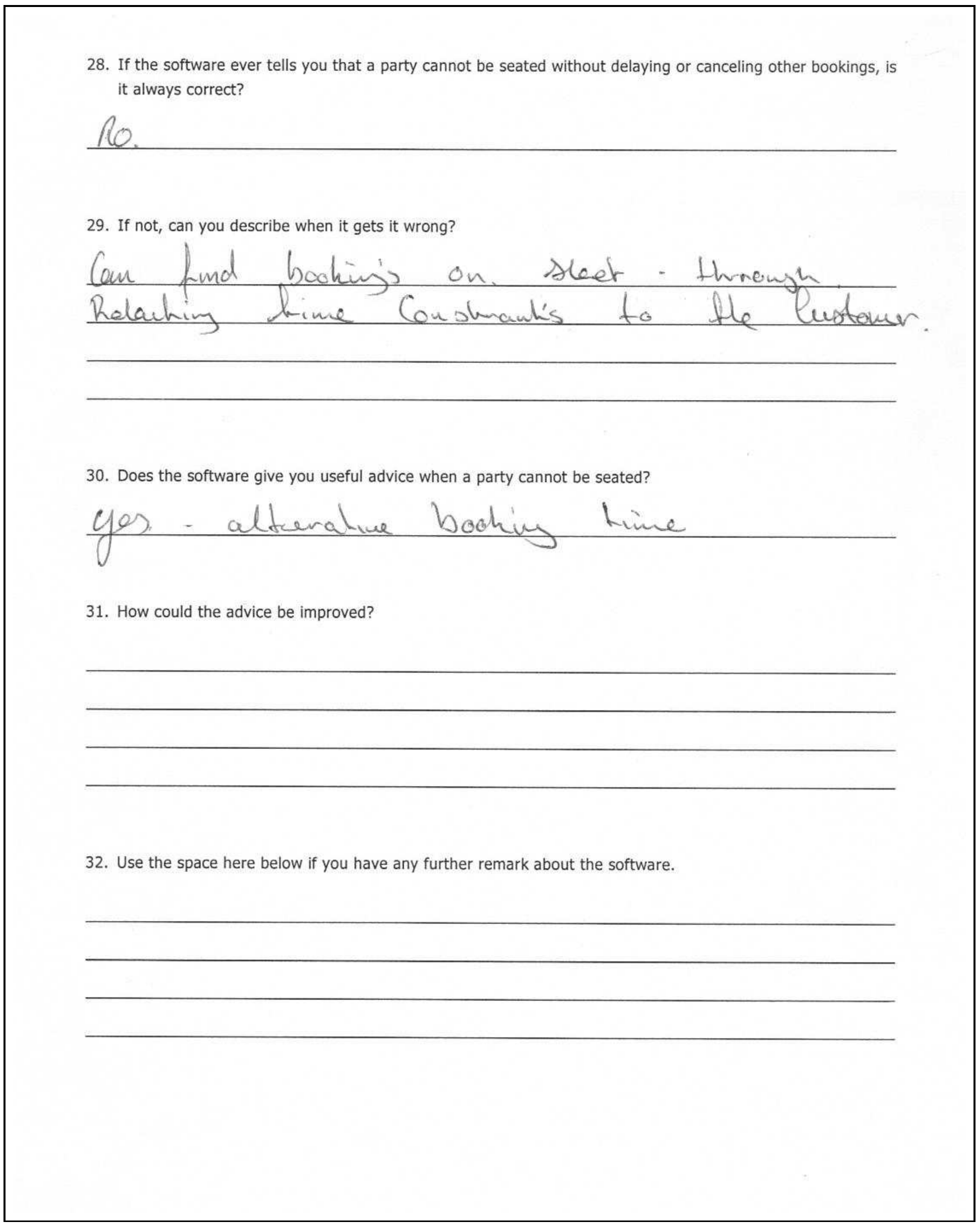
25. How could it be improved?

26. Does the software provide useful advice when a seating plan has to be reconfigured? E.g. if a party is late in arriving, or a party takes longer than expected?

yes.

27. When a seating plan has to be reconfigured, does the software manage to keep most of the seating plan as it was?

yes.



28. If the software ever tells you that a party cannot be seated without delaying or canceling other bookings, is it always correct?

No.

29. If not, can you describe when it gets it wrong?

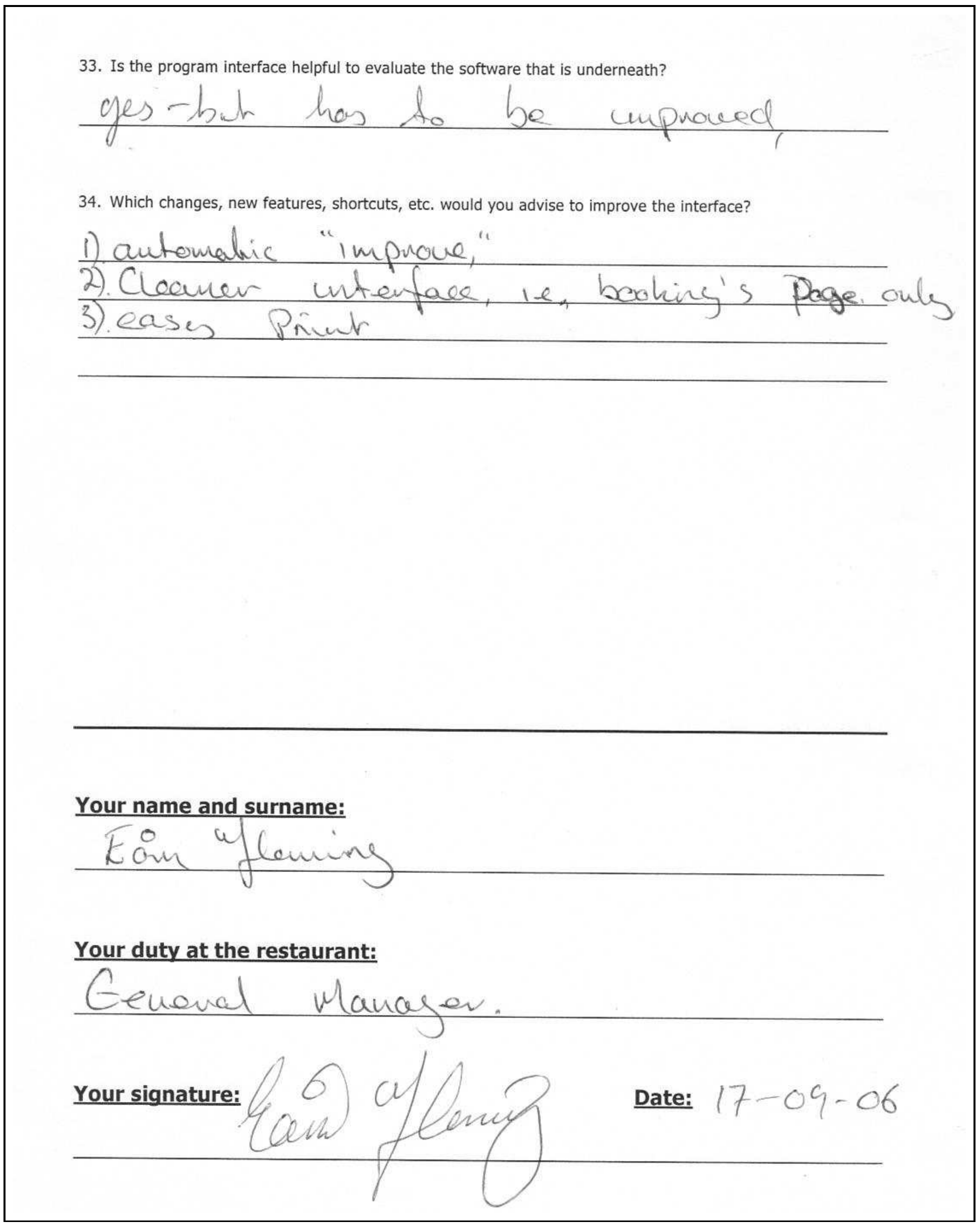
Can find booking's on sheet - through. Relaxing time Constraint's to the Customer.

30. Does the software give you useful advice when a party cannot be seated?

Yes - alternative booking time

31. How could the advice be improved?

32. Use the space here below if you have any further remark about the software.



33. Is the program interface helpful to evaluate the software that is underneath?

yes - but has to be improved,

34. Which changes, new features, shortcuts, etc. would you advise to improve the interface?

1). automatic "improve,"
2). Cleaner interface, i.e. booking's Page. only
3). easier Print

Your name and surname:

Eoin Fleming

Your duty at the restaurant:

General Manager.

Your signature: [illegible] **Date:** 17-09-06

Bibliography

[1] D. Achlioptas, C.P. Gomes, H. Kautz, and B. Selman. Generating satisfiable problem instances. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-00)*, pages 256–261, Austin, TX, USA, 2000. AAAI Press / The MIT Press.

[2] E.M. Arkin and E.B. Silverberg. Scheduling jobs with fixed start and end times. *Discrete Applied Mathematics*, 18:1–8, 1987.

[3] P. Baptiste, P. Laborie, C. Le Pape, and W. Nuijten. *Constraint-Based Scheduling and Planning*, chapter 22, pages 761–760. Handbook of constraint programming. Elsevier, (eds. Rossi, F., van Beek, P. and Walsh, T.), 2006.

[4] J.C. Beck and M.S. Fox. Dynamic problem structure analysis as a basis for constraint-directed scheduling heuristics. *Artificial Intelligence*, 117(1):31–38, 2000.

[5] J.C. Beck and L. Perron. Discrepancy bounded depth first search. In *Proceedings of 2nd International Workshop on Integration of AI and OR Technologies for Combinatorial Optimization Problems (CP-AI-OR-00)*, pages 7–17, Paderborn, Germany, 2000.

[6] J.C. Beck, P. Prosser, and R.J. Wallace. Failing first: An update. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-04), Short Paper*, pages 959–960, Valencia, Spain, 2004.

[7] J.C. Beck, P. Prosser, and R.J. Wallace. Variable ordering heuristics show promise. In *Proceedings of the 10th International Conference on Principles*

and Practice of Constraint Programming (CP-04), pages 711–715, Toronto, Canada, 2004.

[8] J.C. Beck, P. Prosser, and R.J. Wallace. Trying again to fail-first. In *Recent Advances in Constraints. Lecture Notes in Artificial Intelligence. Paper from the Joint Annual Workshop of ERCIM/CoLogNet on Constraint Solving and Constraint Logic Programming (CSCLP-04)*, pages 41–55, volume 3419, Springer, Berlin, 2005.

[9] T. Benoist, E. Bourreau, Y. Caseau, and B. Rotterbourg. Towards stochastic constraint programming: A study of on-line multi-choice knapsack with deadlines. In *Proceeding of the 7th International Conference on Principles and Practice of Constraint Programming (CP-01)*, pages 61–76, Paphos, Cyprus, 2001.

[10] R. Bent and P. Van Hentenryck. Regrets only! Online stochastic optimization under time constraints. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04)*, pages 501–506, San Jose, California, 2004.

[11] R. Bent and P. Van Hentenryck. The value of consensus in online stochastic scheduling. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS-04)*, pages 219–226, Whistler, British Columbia, Canada, 2004.

[12] Russell W. Bent and Pascal Van Hentenryck. Scenario-based planning for partially dynamic vehicle routing with stochastic customers. *Operations Research*, 52(6):977–987, 2004.

[13] D. Bertsimas and R. Shioda. Restaurant revenue management. *Operations Research*, 51(3):473–486, 2003.

[14] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence.*, 65(1):179–190, 1994.

[15] C. Bessière, P. Meseguer, E.C. Freuder, and J. Larrosa. On forward checking for non-binary constraint satisfaction. In *Proceedings of the 5th Inter-*

national Conference on Principles and Practice of Constraint Programming (CP-99), pages 88–102, Alexandria, VA, USA, 1999.

[16] C. Bessière and J.-C. Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings of the 2nd International Conference on Principles and Practice of Constraint Programming (CP-96)*, pages 61–75, Cambridge, Massachusetts, USA, 1996.

[17] C. Bessière and J.-C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 309–315, Seattle, Washington, USA, 2001.

[18] S. Bistarelli, H. Fargier, U. Montanari, F. Rossi, T. Schiex, and G. Verfaillie. Semiring-based CSPs and valued CSPs: Frameworks, properties, and comparison. *Constraints*, 4(3):275–316, 1999.

[19] F. Bonomo and M. Cecowski. Between coloring and list-coloring: μ-coloring. *Electronic Notes in Discrete Mathematics*, 19:117–123, 2005.

[20] F. Bonomo, G. Durán, and J. Marenco. Exploring the complexity boundary between coloring and list-coloring. *Electronic Notes in Discrete Mathematics*, 25:41–47, 2006.

[21] F. Boussemmart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-04)*, pages 146–150, Valencia, Spain, 2004.

[22] J. Branke and D.C. Mattfeld. Anticipatory scheduling for dynamic job shop problems. In *Workshop Proceedings on Online Planning and Scheduling (in AIPS-02)*, pages 3–10, Toulouse, France, 2002.

[23] D. Brélaz. New methods to color the vertices of a graph. *Communications of the Internationally Acknowledged Premier Magazine of the Computing Field (ACM)*, 22(4):251–256, 1979.

[24] K.N. Brown and I. Miguel. *Uncertainty and Change*, chapter 21, pages 731–760. Handbook of constraint programming. Elsevier, (eds. Rossi, F., van Beek, P. and Walsh, T.), 2006.

[25] A. Cesta and R. Rasconi. Execution monitoring and schedule revision for O-Oscar: A preliminary report. In *Workshop Proceedings on Online-03: International Workshop on Online Constraint Solving - Handling Change and Uncertainty (in CP-03)*, pages 9–23, Kinsale, Ireland, 2003.

[26] H.S. Chang, R. Givan, and E.K.P. Chong. On-line scheduling via sampling. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (AIPS-00)*, pages 62–71, Breckenridge, CO, USA, 2000.

[27] H. Chen, C. Gomes, and B. Selman. Formal models of heavy-tailed behavior in combinatorial search. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP-01)*, pages 408–422, Paphos, Cyprus, 2001.

[28] B.M.W. Cheng, J.H.M. Lee, and J.C.K. Wu. Speeding up constraint propagation by redundant modeling. In *Proceedings of the 2nd International Conference on Principles and Practice of Constraint Programming (CP-96)*, pages 91–103, Cambridge, Massachusetts, USA, 1996.

[29] A.C. Davenport and J.C. Beck. A survey of techniques for scheduling with uncertainty. In *unpublished manuscript*, available from <http://tidel.mie.utoronto.ca/publications.php>, 2000.

[30] A.C. Davenport, C. Gefflot, and J.C. Beck. Slack-based techniques for robust schedules. In *Proceedings of the 6th European Conference on Planning (ECP-01)*, pages 7–18, Toledo, Spain, 2001.

[31] R. Debruyne and C. Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, volume 1, pages 412–417, Nagoya, Japan, 1997. Morgan Kaufmann.

[32] Designed and constructed by: Numt.com. "Eco restaurant homepage". <http://www.eco.ie>, (June 2007).

[33] R. Detcher. *Constraint processing*. Morgan Kaufman, 2003.

[34] R. Detcher and A. Detcher. Belief maintenance in dynamic constraint networks. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI-88)*, pages 37–42, Saint Paul, Minnesota, 1988.

[35] M. Drummond, J. Bresina, and K. Swanson. Just-in-case scheduling. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 1098–1104, Seattle, Washington, 1994.

[36] H. Fargier, J. Lang, R. Martin-Clouaire, and T. Schiex. A constraint satisfaction framework for decision under uncertainty. In *Proceedings of the 11th International Conference on Uncertainty in Artificial Intelligence*, pages 167–174, Montreal, Canada, 1995.

[37] H. Fargier, J. Lang, and T. Schiex. Mixed constraint satisfaction: A framework for decision problems under incomplete knowledge. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 175–180, Portland, OR, USA, 1996.

[38] D.W. Fowler and K.N. Brown. Branching constraint satisfaction problems and markov decision problems compared. *Annals of Operations Research*, 118(1-4):85–100, 2003.

[39] E.C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the Association for Computing Machinery*, 32(4):755–761, 1985.

[40] E.C. Freuder and R. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

[41] D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 572–578, Montreal, Canada, 1995.

[42] M.R. Garey and D.S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W.H. Freeman and Company, New York, 1979.

[43] M.R. Garey, D.S. Johnson, G.L. Miller, and C.H. Papadimitriou. The complexity of coloring circular arcs and chords. *Society for Industrial and Applied Mathematics (SIAM) Journal on Algebraic and Discrete Methods*, 1(2):216–227, 1980.

[44] I.P. Gent. A symmetry breaking constraint for indistinguishable values. In *Proceedings of SymCon'01, the CP'01 Workshop on Symmetry in Constraints*, pages 469–473, Paphos, Cyprus, 2001.

[45] I.P. Gent, E. MacIntyre, P. Prosser, B.M. Smith, and T. Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. Division of Artificial Intelligence Technical Report 96-05, University of Leeds, 1996.

[46] I.P. Gent, E. MacIntyre, P. Prosser, and T. Walsh. The constrainedness of search. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 246–252, Portland, OR, USA, 1996.

[47] L. Getoor, G. Ottosson, M. Fromherz, and B. Carlson. Effective redundant constraints for online scheduling. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pages 302–307, Providence, Rhode Island, 1997. AAAI Press / MIT Press.

[48] C.P. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001.

[49] C.P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 431–437, Madison, Wisconsin, USA, 1998.

[50] C.P. Gomes, B. Selman, K. McAloon, and C. Tretkoff. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In *Proceedings of the 4th International Conference on Ar-*

tificial Intelligence Planning Systems (AIPS-98), pages 208–213, Carnegie Mellon University, Pittsburgh Pennsylvania, USA, 1998.

[51] C.P. Gomes and D.B. Shmoys. Approximations and randomization to boost CSP techniques. *Annals of Operations Research*, 130:117–141, 2004.

[52] M. Grotschel, L. Lovasz, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1:169–197, 1981.

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

[54] W.D. Harvey. *Nonsystematic backtracking search.* Phd thesis, Stanford university, Department of computer science, 1995.

[55] E. Hebrard, B. Hnich, and T. Walsh. Robust solutions for constraint satisfaction and optimization. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-04)*, pages 186–190, Valencia, Spain, 2004.

[56] B. Hnich, B.M. Smith, and T. Walsh. Dual modelling of permutation and injection problems. *Journal of Artificial Intelligence Research*, 21:357–391, 2004.

[57] A. Holland and B. O'Sullivan. Weighted super solutions for constraint programs. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-05)*, pages 378–383, Pittsburgh, Pennsylvania, 2005.

[58] H.H. Hoos and E. Tsang. *Local search methods*, chapter 5, pages 135–167. Handbook of constraint programming. Elsevier, (eds. Rossi, F., van Beek, P. and Walsh, T.), 2006.

[59] T. Hulubei and B. O'Sullivan. Optimal refutations for constraint satisfaction problems. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 163–168, Edinburgh, Scotland, 2005.

[60] iSeatz.com. "iSeatz homepage". <*http://www.iseatz.com*>, (June 2007).

[61] S. Kimes. Implementing restaurant revenue management. *Cornell Hotel and Restaurant Administration Quarterly*, 34(3):16–21, 1999.

[62] S. Kimes. Restaurant revenue management. Cornell Hospitality Report 4(2), Center for Hospitality Research at Cornell University, 2004.

[63] R. Korf. Improved limited discrepancy search. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 286–291, Portland, OR, USA, 1996.

[64] R.E. Korf. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[65] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[66] A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25(1):65–74, 1985.

[67] S. Manandhar, A. Tarim, and T. Walsh. Scenario-based stochastic constraint programming. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 257–262, Acapulco, Mexico, 2003.

[68] D. Marx. Precoloring extension on unit interval graphs. *Discrete Applied Mathematics*, 154(6):995–1002, 2006.

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

[70] R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings of the 8th European Conference on Artificial Intelligence (ECAI-88)*, pages 651–656, Munich, Germany, 1988.

[71] A.K. Mok and M.L. Dertouzos. Multiprocessor scheduling in a hard real-time environment. In *Proceedings of the 7th IEEE Texas Conference on Computing Systems*, pages 5–1–5–12, Houston, Texas, 1978.

[72] P. Morris, N. Muscettola, and T. Vidal. Dynamic control of plans with temporal uncertainty. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 494–502, Seattle, Washington, USA, 2001.

[73] OpenTable. "Open Table homepage". <*http://www.opentable.com*>, (June 2007).

[74] A. Petcu and B. Faltings. Optimal solution stability in continuous-time optimization. In *Proceedings of the 6th International Workshop on Distributed Constraint Reasoning (DCR-05)*, pages 207–221, Edinburgh, Scotland, 2005.

[75] T. Petit, J.C. Régin, and C. Bessière. Range-based algorithm for max-csp. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP-02)*, pages 280–294, Ithaca, NY, USA, 2002.

[76] M.L. Pinedo. *Planning and scheduling in manufacturing and services*. Springer Series in Operations Research. Springer, 2005.

[77] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[78] J.F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems (ISMIS-93)*, pages 350–361, Trondheim, Norway, 1993.

[79] Y. Ran, N. Roos, and J. Van Den Herik. Approaches to find near-minimal change solution for dynamic CSPs. In *Proceedings of the 4th International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (in CP-AI-OR-02)*, pages 373–387, Le Croisic, France, 2002.

[80] J.-C. Régin. A filtering algorithm for constraints of difference in csps. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 362–367, Seattle, Washington, 1994.

[81] F. Rossi, P. van Beek, and T. Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[82] Z. Ruttkay. Fuzzy constraint satisfaction. In *Proceedings of the 1st IEEE Conference on Evolutionary Computing*, pages 542–547, Orlando, Florida, USA, 1994.

[83] ILOG SA. Ilog solver 5.3 user's manual. 2003.

[84] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI-94)*, pages 125–129, Amsterdam, The Netherlands, 1994.

[85] T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal on Artificial Intelligence Tools (IJAIT-94)*, 3(2):187–207, 1994.

[86] B.M. Smith. Succeed first or fail-first: A case study in variable and value ordering heuristics. In *Proceedings of the 4th International Conference on Parallel Computing Technologies (PaCT-97)*, pages 321–330, Yaroslavl, Russia, 1997.

[87] B.M. Smith and S.A. Grant. Trying harder to fail first. In *Proceedings of 13th European Conference on Artificial Intelligence (ECAI-98)*, pages 249–253, Brighton, UK, 1998.

[88] B.M. Smith, K. Stergiou, and T. Walsh. Using auxiliary variables and implied constraints to model non-binary problems. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-00)*, pages 182–187, Austin, Texas, 2000.

[89] J.A. Stankovic, M.R. Spuri, K. Ramamritham, and G.C. Buttazzo. *Deadline scheduling for real-time systems*. Kluwer Academic Publisher Boston, 1998.

[90] G.M. Thompson. Dedicated or combinable? A simulation to determine optimal restaurant table configuration. Cornell Hospitality Report, Center for Hospitality Research at Cornell University, 2003.

[91] G. Verfaillie and T. Schiex. Solution reuse in dynamic constraint satisfaction problems. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 307–312, Seattle, Washington, 1994.

[92] T. Vidal and H. Fargier. Handling contingency in temporal constraint networks: From consistency to controllabilities. *Journal of Experimental and Theoretical Artificial Intelligence (JETAI)*, (1):23–45, 1999.

[93] R.J. Wallace and E.C. Freuder. Anytime algorithms for constraint satisfaction and sat problems. *Association for Computing Machinery (ACM) Special Interest Group on Artificial Intelligence (SIGART) Bulletin*, 7(2):7–10, 1996.

[94] T. Walsh. Depth-bounded discrepancy search. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1388–1395, Nagoya, Japan, 1997.

[95] T. Walsh. Search in a small world. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 1172–1177, Stockholm, Sweden, 1999.