TITLE: Implementation and performance evaluation of an online ordering web application for restaurants

DEGREE: Master's degree in Applied Telecommunications and Engineering Management (MASTEAM)

AUTHOR: Ivan BalboteoToledano

TUTOR: Davide Vega d'Aurelio

DATE: January 11, 2016
Overview

QPides! is an application whose main target is to satisfy the need of a more fluid and detailed information between restaurants and their clients. This information includes the restaurant’s location, table allocation, availability, menu content and the customers' comments. Additionally, provides a reliable and secure channel to submit customized orders and payments. This enables to reduce the workload on restaurant staff and invest such resources in maximizing the quality of the service provided.

One of the most important distinguishing features of QPides! application is the table allocation process, since an effective table allocation can be crucial to a restaurant's profitability. Inefficient use of tables means that the restaurant is losing potential customers, but overbooking means that customers are delayed or feel cramped, and so are unlikely to return. In addition, customer behavior is dynamic, and so table allocation should be flexible or quickly reconfigurable, to avoid delays. Restaurant table allocation could be improved if the software can be used by staff with less expertise and knowledge, and that can help the automation and optimization of the allocation process. Specially, for these reasons the idea of QPides! was proposed.

This work aims to develop the backend of QPides! focusing on table allocation problem. Our proposed solution involves designing different table-allocation algorithms. These algorithms solve a constraint satisfaction problem, looking the best combination of tables at any given time that maximize the occupancy rate. The web environment used, in which the algorithms are implemented, comprises Meteor, an open-source JavaScript web application framework optimized for real-time apps, and MongoDB database. The main objective is to improve restaurants performance in terms of occupancy rate and response time.

Evaluation of QPides! performance is done by applying load testing for each table allocation algorithm. All testing carried out in this work are based on dummy data that try to simulate real-world scenarios. The results obtained in the experimental measurements prove that searching with backtracking (BT) is much more efficient. Performing a properly pruning of the search tree does not prevent find the best solution and reduces the computational cost.
INDEX

1 INTRODUCTION ................................................................................................................. 2
  1.1 Background .................................................................................................................. 2
  1.2 Motivation ................................................................................................................ 2
  1.3 Goals ............................................................................................................................ 3
  1.4 Overview of the dissertation ....................................................................................... 3

2 RESTAURANT TABLE MANAGEMENT ............................................................................. 4
  2.1 Introduction .................................................................................................................. 4
  2.2 The problem .............................................................................................................. 4
    2.2.1 Table capacity and combinability ................................................................. 5
    2.2.2 Sources of uncertainty ...................................................................................... 7
    2.2.3 The sequential decision problem .................................................................. 9
  2.3 A new service scheduling: QPides! ............................................................................ 10
  2.4 Constraint Programming ......................................................................................... 11
    2.4.1 Solving CSPs using search ............................................................................. 11

3 TECHNOLOGY SELECTION ............................................................................................. 14
  3.1 Introduction ................................................................................................................ 14
  3.2 Programming languages .......................................................................................... 14
    3.2.1 Higher Levels of Languages ........................................................................ 15
    3.2.2 The Web ....................................................................................................... 15
    3.2.3 Benchmark testing ....................................................................................... 17
  3.3 Web framework ........................................................................................................ 18
    3.3.1 MVC ............................................................................................................ 18
    3.3.2 Framework ................................................................................................... 20
    3.3.3 Features comparison ................................................................................. 20
    3.3.4 Selected MV* framework ....................................................................... 22

4 QPIDES! IMPLEMENTATION ON A WEB PLATFORM ................................................. 25
  4.1 Table allocation algorithms ....................................................................................... 25
    4.1.1 Basic ............................................................................................................. 26
    4.1.2 Random .......................................................................................................... 28
    4.1.3 Backtracking .................................................................................................. 28
    4.1.4 MongoDB Limits ....................................................................................... 32
  4.2 Use case analysis ..................................................................................................... 33

5 EXPERIMENTAL MEASUREMENTS ......................................................................... 39
  5.1 Introduction ................................................................................................................. 39
5.2 Defining the test scenario ........................................................................................................39
5.3 Load testing ............................................................................................................................40
5.4 Results ..................................................................................................................................41
  5.4.1 Average time of reservation acceptance ........................................................................41
  5.4.2 Reservations denied .........................................................................................................42
  5.4.3 Occupancy rate ................................................................................................................43
  5.4.4 Resources consumption ....................................................................................................44

6 CONCLUSIONS ......................................................................................................................47
6.1 Environmental study ..............................................................................................................47

7 FUTURE RESEARCH ..............................................................................................................48

8 BIBLIOGRAPHY .......................................................................................................................49
INDEX OF FIGURES

Figure 2-1 Restaurant table management: the sequential decision problem...... 9
Figure 2-2 Search tree for the CSP .................................................. 12
Figure 3-1 What programming language should you learn? ..................... 14
Figure 3-2 .NET vs. Node.js performance comparison .......................... 18
Figure 3-3 Elements of MVC design pattern .................................... 19
Figure 4-1 Basic table-allocation algorithm ..................................... 27
Figure 4-2 Procedure of BT method for finding solutions ....................... 29
Figure 4-3 Flow diagram of backtracking algorithm ............................ 31
Figure 4-4 Chef use case (left) and Restaurant Manager use case (right) ... 34
Figure 4-5 Hostess use case (left) and Administrator use case (right) ...... 35
Figure 4-6 Consumer use case .......................................................... 36
Figure 5-1 Kadira.io performance monitoring system ............................. 39
Figure 5-2 Average time of reservation acceptance .............................. 41
Figure 5-3 Reservations denied ....................................................... 42
Figure 5-4 Occupancy rate ............................................................... 44
Figure 5-5 RAM usage ................................................................. 45
Figure 5-6 CPU usage ..................................................................... 46

INDEX OF TABLES

Table 2-1 Problem instance .................................................................. 6
Table 2-2 Table allocation plan 1 .................................................... 6
Table 2-3 Table allocation plan 2 .................................................. 6
Table 2-4 Main sources of uncertainty .......................................... 7
Table 2-5 A description of the main events that can occur in restaurant table management. ....................................................... 9
Table 3-1 Programming languages comparison .................................. 17
Table 3-2 Framework comparison ............................................... 21
Table 4-1 Use case grid ................................................................. 38
Table 4-2 Use case analysis template ............................................. 38
LIST OF ACRONYMS

NFC: Near Field Communication (RF technology)
GPS: Global Positioning System
IP: Internet Provider
HLL: High Level Language
IDE: Integrated Development Environment
HTML: HyperText Markup Language (and file extension)
CSS: Cascading Style Sheet
PHP: Personal Home Page
DOM: Document Object Model
IO: Input/Output
DRY: Don’t Repeat Yourself.
MVC: Model View Controller
MVP: Model View Presenter
MVVP: Model-View-ViewPresenter
UI: User Interface
REST: Representational State Transfer
IDevE: Isomorphic Development Ecosystem
API: Application Program Interface
NPM: Node Package Manager
UML: Unified Modeling Language
URL: Uniform Resource Locator
SQL: Structured Query Language
ID: Identifier
JSON: JavaScript Object Notation
BT: Backtracking
1 Introduction

1.1 Background
According to a report from the National Restaurant Association (NRA), a significant numbers of consumers, especially those under 45 years old, reported that they would use new technologies to view menus, order meals and make reservations. Hudson Riehle, Senior Vice President of the NRA’s Research and Information Division, emphasizes the importance of technology in not only capturing this younger demographic, but also in improving the restaurant’s operations [1].

There is the application market plenty of restoration solutions aimed to specific independent tasks, e.g. booking, ordering or payment. There is not, in most of the cases, no possible communication between these applications, support for dynamic table reallocation or even several replicas of the same application customized for different providers.

QPides! would add extra features to the existing ones in the market to make an all-in-one application. The consumer would only require a single application, where as the restaurant owner would implement a business solution customized to their everyday business.

1.2 Motivation
Effective table management can be crucial to restaurant’s profitability. Inefficient use of tables means that the restaurant is losing potential customers, but overbooking means that customers are delayed or feel cramped, 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.

Some companies already take into account some of these problems and propose reservation systems that help manage the restaurant. For example, the restaurant reservation system by SeatMe includes a table and wait list management software. [2]

Table allocation is a constraint satisfaction problem. Restaurants must manage reservations, and manage unexpected events in real-time, making best use of available resources, and providing a superior quality service to consumers [3]. Specially, for these reasons the idea of QPides! was proposed.

QPides! is an infrastructure aimed to facilitate the reservation, selection and acquisition of products or services from the client’s customer wish until its delivery. The infrastructure provides the customer a unique solution:

- Geo-localized advertisement for possible consumers
- Targeted customer offers
- Manage menu/catalog of the services/products offered
• Control product availability
• **Table allocation**
• Smart and secure order payment management (Web, NFC, etc.)
• Business data mining:
  o consumer preferences/suggestions
  o service/delivery time history
  o best-selling product, ...

In this thesis, we focus on the development of a QPides component: the table allocation service. The solution is based on constraint programming over a web environment, and handles both flexibility and stability. It is programmed using a JavaScript web framework, called Meteor, and MongoDB database. The research underneath this thesis was carried out using information from a virtual restaurant, was tested using a real server, and was finally validated by running a load and performance web testing.

### 1.3 Goals

The goals of this dissertation are:

1) Design table allocation algorithms that maximize the restaurant occupancy rate.

2) Evaluate and compare the performance of the algorithms in terms of restaurant occupancy rate and response time in a web environment.

3) Implement the QPides system on a web platform.

### 1.4 Overview of the dissertation

The structure of the dissertation is as outlined below.

Section 2 presents more details of the table management problem and reviews the elements of constraint programming. Section 3 describes the working environment, detailing the hardware platform and software tools that have been used to carry out the implementation of QPides web service, while section 4 presents a use cases analysis of our software. Section 5 shows the results of performance testing. Finally, section 6 and 7 describes conclusions and future work, respectively. All source code used in this work is included in the annexes.
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 and high dynamism, and describing the classical solutions.

First, we divide the problem into two phases: booking and floor management. Then, we present the main elements 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.

Finally, we represent table management as a sequential decision problem and introduce the basics of constraint programming. A complete and detailed description of restaurant table management can be found in [3]. However, in order to make this thesis a self-contained document this section provides a brief description of such management.

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. If a request cannot be accommodated on the current booking sheet, the customer must be persuaded to accept another time or the request must be denied. 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 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, e.g. 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 day.
2.2.1 Table capacity and combinability

Commonly, restaurants have sets of tables of different capacities. Customers 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 tables may be combinable with others. There may be different possible layouts (or restaurant configurations) a restaurant can assume, depending on how tables are combined.

We model table allocation as a scheduling problem, viewing tables as resources, and reservations as tasks. Each reservation has a fixed start and end time, and a size. Each reservation must be allocated to a table (or set of tables), such that the table is large enough for the reservation, and such that no two reservations that overlap in time are allocated to the same table. The problem is to determine whether a set of reservations can be allocated, and to provide a feasible allocating plan if there is one.

Table 2-1 shows a problem instance with five reservations. Table 2-2 and Table 2-3 show possible allocation plans. The Table 2-1 contains the following columns: Reservation, Size, Start and End. The Reservation column holds the identifier of reservations: R1, R2, R3, R4, R5. The Size column holds the number of people in each reservation. Start and End columns specify, respectively, the time slot in which each reservation starts and ends.

Given this problem instance, suppose you are in the time slot number 0 and all the tables are available. The first reservation for that time slot is R1 with a required capacity of 2 people. The T1 table is the first available table whose capacity is at least equal to that required for the R1 reservation. Then, the T1 table is allocated to R1 for time slots number 0 and 1.

The R2 reservation also starts in the time slot number 0. R2 requires a capacity for 4 people. One possible solution is to combine the tables T2 and T3 to serve up to 6 people. Thus, T2 and T3 tables are allocated to R2 for time slots 0 and 1.

R3 reservation starts in the time slot 1. At that time slot, the tables T1, T2 and T3 are occupied. So the only table available is T4. As the capacity of this table is greater than that required by the reservation R3, T4 is assigned to this reservation until the time slot number 2.

In the time slot number 2 are two reservations: R4 and R5. Each with a required capacity of 2 people. In that time slot are available all the tables except the T4. The T1 table is the first available table whose capacity is at least equal to that required for the R4 reservation. Then, the T1 table is allocated to R4 for time slots number 2 and 3.

The T2 and T3 tables are available for allocation to the last reservation, R5. Both tables have a capacity of 3 people and the capacity required for the reservation is 2. Therefore, the allocation of a table or another is irrelevant. So, T2 is assigned to the R5 reservation until the time slot number 3.
Another possible table allocation plan is shown in Table 2-3. For this instance of the problem, there are multiple valid solutions. However, not all offer the same performance. For example in terms of occupancy rate is concerned. It is therefore necessary to apply a method for finding the best solution to the problem by adding new constraints. Section 4.1.3 explains the designed algorithms to maximize the occupancy rate.

<table>
<thead>
<tr>
<th>Reservation</th>
<th>Size</th>
<th>Start</th>
<th>End</th>
<th>Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>?</td>
</tr>
<tr>
<td>R2</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>?</td>
</tr>
<tr>
<td>R3</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>?</td>
</tr>
<tr>
<td>R4</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>?</td>
</tr>
<tr>
<td>R5</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>?</td>
</tr>
</tbody>
</table>

Table 2-1 Problem instance

<table>
<thead>
<tr>
<th>Table[size]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1[2]</td>
<td>R1</td>
<td>R1</td>
<td>R4</td>
<td>R4</td>
</tr>
<tr>
<td>T2[3]</td>
<td>R2</td>
<td>R2</td>
<td>R5</td>
<td>R5</td>
</tr>
<tr>
<td>T3[3]</td>
<td>R2</td>
<td>R2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T4[4]</td>
<td>R3</td>
<td>R3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2-2 Table allocation plan 1

<table>
<thead>
<tr>
<th>Table[size]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1[2]</td>
<td>R1</td>
<td>R1</td>
<td>R5</td>
<td>R5</td>
</tr>
<tr>
<td>T2[3]</td>
<td>R3</td>
<td>R3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3[3]</td>
<td>R4</td>
<td>R4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T4[4]</td>
<td>R2</td>
<td>R2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2-3 Table allocation plan 2

Extending physical constraints with business rules
Table, configuration, and layout capacities represent physical constraints and therefore cannot be violated. Examples of such constraints are:

1. A table for two cannot accommodate four.

2. 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.

3. The number of people eating at the same time cannot exceed the restaurant capacity.

However, by simply satisfying these physical constraints, table allocation may lead to very poor allocations and profit. For example, large unusable time slots between two consecutive dinners on the same table must be avoided. 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 unoccupied seats in oversized tables, or minimizing the time between meals.

### 2.2.2 Sources of uncertainty

Table allocation is a 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 2-4 reports the main sources of uncertainty, concerning customer behavior and restaurant performance.

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

**Table 2-4 Main sources of uncertainty**

#### Table demand

The lack of knowledge in real-world scenarios 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 allocate tables that maximizes the occupancy rate. 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 allocate 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? 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 are 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 eight with a reservation for five at the door, but all the tables that can accommodate eight have already been reserved.

**Kitchen and staff efficiency**
Kitchen and staff have a limited number of resources. Roughly, assuming there is no shortage of staff, a restaurant has the potential to serve a limited number of people at the same time. This also depends on the amount and the type of food that has been ordered. Thus, if for example a restaurant can simultaneously serve at most eight meals, the tendency would be to limit the reservations for the same booking time to eight seaters. 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 is stretched, and therefore some following dinners may be 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).

![Diagram](https://via.placeholder.com/150)

**Figure 2-1 Restaurant table management: the sequential decision problem**

Table 2-5 summarizes the possible events during table management, in booking (B) or in the floor phase (F), and describes the possible decisions for each event type.

<table>
<thead>
<tr>
<th>Possible event</th>
<th>Stage</th>
<th>Possible decisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>New booking request</td>
<td>B, F</td>
<td>If there is a suitable table available in the current table allocation then accept; otherwise suggest alternatives times or reject</td>
</tr>
<tr>
<td>Booking change</td>
<td>B, F</td>
<td>If the change can be accommodated in the current table allocation the accept; otherwise suggest alternatives times or reject</td>
</tr>
<tr>
<td>Booking cancellation</td>
<td>B, F</td>
<td>Remove the booking from the current table allocation</td>
</tr>
<tr>
<td>Walk-in</td>
<td>F</td>
<td>If there is a suitable table available in the current table allocation then accept; otherwise suggest alternatives times or reject</td>
</tr>
<tr>
<td>No-show</td>
<td>F</td>
<td>If a booking does not arrive after 30 minutes then remove it from the current table allocation.</td>
</tr>
<tr>
<td>On-time /late arrival</td>
<td>F</td>
<td>Accommodate the party at the earliest available time.</td>
</tr>
<tr>
<td>Early arrival</td>
<td>F</td>
<td>Accommodate the party at the earliest available time (even earlier than the booking time if this does not increase delays for other bookings).</td>
</tr>
<tr>
<td>Dinner shorter than expected</td>
<td>F</td>
<td>No decision is necessary, though the slot of time that has been gained can now be used for future allocations.</td>
</tr>
<tr>
<td>Dinner longer than expected</td>
<td>F</td>
<td>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.</td>
</tr>
<tr>
<td>Party turning up with fewer people than the table size that was booked</td>
<td>F</td>
<td>Try to reallocate the party to a table of more suitable capacity; otherwise seat the party at the table originally assigned.</td>
</tr>
<tr>
<td>Party turning up with more people than the table size that was booked</td>
<td>F</td>
<td>If the preassigned table is no longer of suitable capacity then accommodate the party to a suitable as soon as one becomes available.</td>
</tr>
<tr>
<td>Party that arrives believing a booking has been made when none has been recorded</td>
<td>F</td>
<td>Accommodate the party at the earliest available time.</td>
</tr>
</tbody>
</table>

**Table 2-5 A description of the main events that can occur in restaurant table management.**
2.3 A new service scheduling: QPides!

Although the use of intelligent automation to real world problems has shown a considerable and increasing success in recent years [4], the problem of restaurant table allocation has been left largely unexplored if we compare it for example with guiding autonomous vehicles and advanced robots.

QPides! table allocation was created with the purpose to make efficient use of tables in a restaurant, with an optimal solution that maximizes the occupancy rate. This not only improves the profitability of the restaurant but also the customer experience. In addition to this functionality, the application has other features that are listed below.

**Geo-localized advertisement for possible consumers**

Typically, customers are looking for nearby restaurants to enjoy the experience. QPides! devotes a section to offer discounts from the nearest restaurants using geolocation, either by GPS or by IP addresses. So that at a glance we have access to a wide range of options within minutes of our current position.

**Targeted customer offers**

Restaurant sector customers can be grouped according to the desired characteristics of the service at that time. For example, a romantic setting for couples or a birthday celebration. This achieves direct offers to potentially interested customers.

**Manage menu/catalog of the services/products offered**

The QPides! system incorporates a digital menu, customizable by the restaurant, which includes information regarding products or services offered. In this information, it is a product description, ingredients contained in it, pictures and price. This feature allows restaurants to always keep an updated menu and avoid showing products already have been removed, thereby improving customer satisfaction.

**Control product availability**

The integration of the application in the kitchen is another important factor for managing the availability of products in real time. When a product runs out temporarily disappears from the digital card and it is added to the list of sold out products. Thus, the product availability is monitored and allows managing more efficiently the purchasing tasks.

**Smart and secure order payment management (Web, NFC, etc.)**

The QPides! system has a smart payment system that allows payment from the client terminal itself. Thus, it is not necessary to wait for the restaurant to bring the receipt and the client can get a proof if the transaction was successful. From the point of view of the restaurant, it is possible to release the table before and thus increases the efficiency in rotating tables.

**Business data mining:**
All information related to QPides system is stored in a database. This allows access to data such as customer preferences or suggestions, record delivery times or the best selling product of the month. All this information is analyzed in detail in order to continuously improve restaurant operations.

2.4 Constraint Programming

Constraint programming is the study of computational systems based on constraints. The idea of constraint programming is to solve problems by stating constraints (conditions, properties) which must be satisfied by the solution.

Work in this area can be tracked back to research in Artificial Intelligence in the sixties and seventies. Only in the last decade, however, has there emerged a growing realization that these ideas provide the basis for a powerful approach to programming, modeling and problem solving and that different efforts to exploit these ideas can be unified under a common conceptual and practical framework, constraint programming [5].

Constraint programming uses an approach to solve the problem called Constraint Satisfaction Problem (CSP). This is the approach that has been used in the thesis to solve the table allocation problem, considering that the goal is to maximize the occupancy rate in restaurants.

Constraint Satisfaction arose from the research in Artificial Intelligence (combinatorial problems, search) [5]. The Constraint Satisfaction Problem (CSP) is a problem where one is given:

- a finite set of variables, e.g. restaurant tables.
- a function which maps every variable to a finite domain, e.g. table capacity
- a finite set of constraints, e.g. total capacity of allocated tables greater than required capacity by consumer.

Each constraint restricts the combination of values that a set of variables may take simultaneously. A solution of a CSP is an assignment to each variable a value from its domain satisfying all the constraints. The task is to find one solution or all solutions. Thus, the CSP is a combinatorial problem that can be solved by search [5].

2.4.1 Solving CSPs using search

A trivial algorithm solves such problems or finds that there is no solution. This algorithm generates all possible combinations of values and, then, it tests whether the given combination of values satisfies all constraints or not (consequently, this algorithm is called generate and test). Clearly, this algorithm takes a long time to run so the research in the area of constraint satisfaction
Restaurant table management

concentrate on finding algorithms which solve the problem more efficiently, at least for a given subset of problems [5].

A more efficient method uses the depth-first search, typically called backtracking (BT) that is the most common algorithm for performing systematic search [6].

The search problem can be defined as follows [6]:

- The nodes are assignments of values to some subset of the variables.
- The neighbors of a node $N$ are obtained by selecting a variable $V$ that is not assigned in node $N$.
- The start node is the empty assignment that does not assign a value to any variables.
- A goal node is a node that assigns a value to every variable.

In this case, it is not the path that is of interest, but the goal nodes.

For a better understanding of the BT method, the following example is proposed. Supposing there is a CSP with the variables $A$, $B$, and $C$, each with domain $\{1,2,3,4\}$. The constraints are $A<B$ and $B<C$. A possible search tree is shown in Figure 2-2.

![Search tree for the CSP](image)

In this figure, a node corresponds to all of the assignments from the root to that node. The potential nodes that are pruned because they violate constraints are labeled with ✗. The leftmost ✗ corresponds to the assignment $A=1$, $B=1$. This violates the $A<B$ constraint, and so it is pruned.

This CSP has four solutions. The leftmost one is $A=1$, $B=2$, $C=3$. The size of the search tree, and thus the efficiency of the algorithm, depends on which variable
is selected at each time. A static ordering, such as always splitting on $A$ then $B$ then $C$, is less efficient than the dynamic ordering used here. The set of answers is the same regardless of the variable ordering.

In the preceding example, there would be $4^2 = 64$ assignments tested in a generate-and-test algorithm. For the BT method, there are 22 assignments generated.

Searching with BT can be much more efficient than generate and test. Generate and test is equivalent to not checking constraints until reaching the leaves. Checking constraints higher in the tree can prune large subtrees that do not have to be searched.
3 Technology selection

3.1 Introduction
There are many programming languages and tools to develop a web solution based on constraint programming. The choice of each of them is not easy and can lead to project failure. In this chapter, we present a comparison between the different programming languages and existing web frameworks to conclude with the selection of the most appropriate.

3.2 Programming languages
In the early days of computing, programming was done in the only language that the microprocessor understood: its own binary code, also called machine language or machine code. According to the evolution of computers over time, this primitive language was replaced by others simpler to learn and more comfortable to use, called “high level language” (HLL). This section discusses the different popular programming languages and help us choose which language fits best with our goals. If we look at the Figure 3-1, extracted from [7], we can see the recommended programming languages depending on the type of application to be developed. The following sections discuss in more detail each of the languages.

Figure 3-1 What programming language should you learn?
3.2.1 Higher Levels of Languages

Today there are hundreds of programming languages, although only few of them are widely used. Even within these high-level languages, some languages (such as Java and C#) are more optimized for the machine while other languages choose greater flexibility for the programmer. In this section, we will focus on the most popular programming languages, showing the basic principles of each one. Information related with this section is extracted from [7].

3.2.1.1 Java and C#

Java and C# are two very similar programming languages that are well optimized and have stricter rules to help prevent programming mistakes. Code in these languages need to be “compiled” into lower-level code before it runs, and all variables need to be “declared” with their name and type. They also enforce/encourage a methodology known as “object-oriented programming”, requiring all code to belong to an “object”.

People who program in these languages use an IDE to write their software in, which can provide various features to help with programming, such as auto-completion suggestions while they code, and automatic highlighting of certain errors. The rules in these languages will help you detect certain errors before you even run your code, which can be especially helpful when learning programming. However, Java or C# are not made for writing simple scripts, and they are not as popular for quickly creating dynamic websites.

Java code is not directly converted into machine code. Instead, it runs on a "virtual machine" which can run on all sorts of different hardware. This lets Java fulfill their slogan of "Write once, run anywhere". It is used in regular desktop applications, but its most popular consumer use is for creating Android apps.

C# can be called "Microsoft's Java", but it also has features missing in Java. It does not yet support multiple platforms as well as Java does, but it does fit well with Microsoft's other offerings. Microsoft has started to open-source C# and associated technologies, so support for other platforms will improve over time. Both languages are popular in large companies, large projects and in projects that need optimized code.

3.2.2 The Web

The "interpreted languages" are more flexible, and are probably a better choice for a beginner who doesn't care about the reasons mentioned above. Since they're popular on the web, we will quickly review how websites work before going through different languages.

When you view a website, a central computer called a server sends you a web page through the Internet. Sometimes, they just send a static page that was sitting there on the server, but on modern sites, the page is often dynamically created for you. That means some code was being run on the server (the “backend”) to generate the page that it sent over to you. Websites can use any
language on their back-end that is supported by their web host. The page that gets displayed is formatted in static HTML (a MARKUP LANGUAGE), but it can contain JavaScript that runs in the browser which allow it to do many more things.

### 3.2.2.1 JavaScript

Since JavaScript runs in all browsers, it can be a good choice of language to learn. No installation is required, since it can immediately be tried out in the browser. JavaScript can be used for visual effects, but also for doing things without having to update the entire webpage. Modern web apps require JavaScript for many of their features.

### 3.2.2.2 PHP

PHP is a language built for creating dynamic web pages, and it runs on the server-side. Let’s say you just finished building a website without programming and now you want to be able to customize things further. A large number of websites and scripts are built using PHP, and web hosts often come with a list of one-click-install scripts. However, PHP has some issues, such as a messy syntax and certain inconsistencies and quirks. This means you should probably learn a different language if you just want to learn programming or you want to create an entirely new web app.

### 3.2.2.3 Python

If you just want an easy and elegant language to learn programming, Python is a good choice. Unlike PHP and JavaScript, which are made for the web, Python is a general-purpose language that is often used outside of websites. Python aims to be very readable, so even a beginner could figure out what some simple Python code accomplishes. Python has the unusual feature of using indentation to mark different parts of code. This makes the code look less cluttered, but can sometimes cause issues when copying code. Python is a good choice to go with if you don’t have a specific goal that fits with one of the other languages.

The Table 3-1 [7] provides some additional info about each language.

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>C#</th>
<th>Python</th>
<th>JavaScript</th>
<th>PHP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used for creating</td>
<td>Android apps, large websites</td>
<td>Windows apps, large websites</td>
<td>Math scripts, websites</td>
<td>Anything that runs on a browser (and beyond)</td>
<td>Applications built on older scripts like Wordpress</td>
</tr>
<tr>
<td>Used specially by</td>
<td>Large companies (Banks, e-commerce, Google, etc)</td>
<td>Large companies (Microsoft, healthcare, etc.)</td>
<td>Academics, startups, Google</td>
<td>All websites</td>
<td>Older companies, startups, Facebook.</td>
</tr>
<tr>
<td>Pro / Unique Feature</td>
<td>Well-optimized Java Virtual Machine for LINQ for easily querying and updating</td>
<td>List comprehensions for creating lists based on other</td>
<td>Only language that runs in browser.</td>
<td>Quick to set up server and web host.</td>
<td></td>
</tr>
</tbody>
</table>
### Technology selection

<table>
<thead>
<tr>
<th>Cons / Tradeoffs</th>
<th>How language is executed</th>
<th>Top Web Framework</th>
<th>Programming languages comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strict rules help catch errors but reduce flexibility and brevity.</td>
<td>Compiled to run on the Java Virtual Machine.</td>
<td>Spring MVC</td>
<td>3.2.3 Benchmark testing</td>
</tr>
<tr>
<td>Poor integration with platforms that do not include Windows.</td>
<td>Compiled to run on the Common Language Infrastructure</td>
<td>ASP.NET MVC</td>
<td>Understanding differences between programming languages is crucial. If wrong language is chosen for a project, it will take a lot of time and efforts to change the course and re-implement the project or its part in different language. This benchmark test is based on the article published on the webpage [8] and is designed to demonstrate the performance difference between two popular programming languages: C# and node.js (JavaScript Framework). However, we re-run the test using a server and client on the same local machine to verify the results.</td>
</tr>
<tr>
<td>Somewhat slow. Using whitespace can cause occasional issues.</td>
<td>Commonly executed on CPython.</td>
<td>Django</td>
<td>If you talk to any modern web developer, today chances are you will hear about node.js. One of the key reasons most argue is that node.js is fast, scalable because of forced non-blocking IO, and its efficient use of a single threaded model. However, I was never really sold into the notion that node.js is supremely fast because there aren’t any context switches and thread synchronizations. If that meant consistently higher performance, then sure, that would make sense. Therefore, I wanted to test this theory and find out exactly how fast node.js was compared to .NET.</td>
</tr>
<tr>
<td>Its flexibility sometimes leads to inconsistencies or bugs.</td>
<td>Runs within any browser from source code.</td>
<td>Node.js for server, many for front-end</td>
<td>The test involves IO (ideally not involving a database) and some computation. Furthermore, it is intended to do this under load, so that we can see how each system behaves under stress condition. The test consists of the following: we have approximately 200 files, each containing somewhere between 10 to 30 thousand random decimals. Each request to the server would contain a number such as: /1 or /120, the service would then open the corresponding file, read the contents, and sort them in memory and output the median value. That’s it. Our goal is to reach a maximum of 200 simultaneous requests, so the idea is that each request would have a corresponding file without ever overlapping.</td>
</tr>
<tr>
<td>Makes it easy to follow bad practices and leave security holes.</td>
<td>Commonly executed on the official Zend PHP implementation.</td>
<td>Laravel, etc.</td>
<td></td>
</tr>
</tbody>
</table>
Then, .NET and Node.JS will create a basic HTTP listener. The plan is to create a simple .NET console app that drives load to both .NET and Node.JS services. The key point here is that we test services using the same client.

The Figure 3-2 shows that node.js wins easily since the response time is lower for any level of concurrency. On average, node.js is 27.74 milliseconds faster for each concurrent request. For our project, the key pivot point is performance, scalability over anything else and node.js clearly shines as we’ve shown in this benchmark test.

3.3 Web framework

When a developer creates a website from scratch, its chances that he has to resolve some of the same problems over and over again. Doing so is tiresome and violates one of the core tenants of good programming, Don’t Repeat Yourself (DRY) [9].

Luckily, other people long ago noticed that web developers face similar problems when building a new site, so they teamed up and created frameworks that provide lot of components ready to use. In short, frameworks exist to avoid having to reinvent the wheel and help alleviate some of the overhead when a new site is built.

3.3.1 MVC

Design patterns are important to write maintainable and reusable code. A pattern is a reusable solution that can be applied to commonly occurring problems in software design [10]. It is highly recommended that developers decouple the app into a series of independent components following the MVC pattern.

MVC offers architectural benefits over standard JavaScript. It can assist programmers to write better organized, and therefore more robust and
maintainable code. This pattern has been used and extensively tested over multiple languages and generations of programmers.

MVC is composed of three components (see Figure 3-3):

1) The **model** represents the data, and does nothing else. The model does NOT depend on the controller or the view.

2) The **view** displays the model data, and sends user actions (e.g. button clicks) to the controller. The view can:
   a. be independent of both the model and the controller; or
   b. actually be the controller, and therefore depend on the model.

3) The **controller** provides model data to the view, and interprets user actions such as button clicks. The controller depends on the view and the model. In some cases, the controller and the view are the same object.

There are a few variations of the MVC design pattern such as MVP (Model–View–Presenter) and MVVP(Model–View–ViewModel). Even with the so-called MVC design pattern itself, there is some variation between the traditional MVC patterns vs. the modern interpretation in various programming languages. For example, some MVC–based frameworks will have the view observe the changes in the models while others will let the controller handle the view update. This section is not focused on the comparison of various implementations but rather on the separation--of--concerns and its importance in writing modern web apps [10].

To summarize, the MVC pattern brings modularity to application developers and it enables:

- Reusable and extendable code.
• Separation of view logic from business logic.

• Allow simultaneous work between developers who are responsible for different components (such as UI layer and core logic).

• Easier to maintain.

3.3.2 Framework
In the last few years, a series of JavaScript MVC frameworks have been developed. However, some are designed to develop only a part of the web application and others to create an entire web.

Creating a website in general involves three factors: the client, the server and the database. The client refers to the software part that interacts with the user and the server refers to the part that processes the input from the client. Such abstraction is important to keep separate the different parts of the system. Framework examples include Backbone.js or Ember.js as client-side and others like Express [11] or Geddy [12] as server-side. Other frameworks such as Meteor [13], a full-stack framework bundled with scaffolding, template engines, websocket and persistence libraries, help us to build full web apps.

While they all have their unique advantages, each one of them follows some form of MVC pattern with the goal of encouraging developers to write more structured JavaScript code. With so many different JavaScript frameworks to choose from, selecting the right one for our project is not an easy task. This chapter aims to analyze those frameworks mentioned above in order to choose the right one among all.

3.3.3 Features comparison
There are really important features a framework should have to provide the necessary foundation to build useful applications. This thesis has conducted a study on the features of today's most popular web frameworks. Relevant information was obtained from [14][15][16]. As the list can be very long, we have prepared a brief summary in a comparative table (see Table 3-2) which includes the most important features for a web framework.

<table>
<thead>
<tr>
<th></th>
<th>Backbone.js</th>
<th>Ember</th>
<th>Express</th>
<th>Geddy</th>
<th>Meteor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework</td>
<td>MVP</td>
<td>MVC</td>
<td>MVC</td>
<td>MVC</td>
<td>MVVM</td>
</tr>
<tr>
<td>Template Support</td>
<td>Yes, underscore.js</td>
<td>Yes, Handlebars</td>
<td>Yes, Jade, Consolidate.js</td>
<td>Yes, Handlebars, EJS, Jade, Mustache</td>
<td>Yes, Spacebars</td>
</tr>
<tr>
<td>Nested Template Support</td>
<td>No</td>
<td>Ember + Handlebar has support</td>
<td>Yes (Jade has support)</td>
<td>Yes (Jade has support)</td>
<td>Yes</td>
</tr>
<tr>
<td>Auto binding</td>
<td>No</td>
<td>Yes (only with Handlebar)</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Below we give an overview for each of the frameworks included in the comparison.

**Backbone.js** is a JavaScript library with a RESTful JSON interface and is based on the model–view–presenter (MVP) application design paradigm. Backbone is known for being lightweight, as its only dependency is on one JavaScript library, Underscore.js. It is designed for developing single-page web applications, and for keeping various parts of Web applications (e.g. multiple clients and the server) synchronized. Backbone was created by Jeremy Ashkenas, who is also known for CoffeeScript.

**Ember.js** is an open-source client-side JavaScript web application framework based on the model-view-controller (MVC) software architectural pattern. It allows developers to create scalable single-page applications by incorporating common idioms and best practices into a framework that provides a rich object model, declarative two-way data binding, computed properties, automatically updating templates powered by Handlebars templating library, and a router for managing application state.

**Express.js** is a lightweight web application framework to help developers to organize the web application into MVC architecture on the server side. Provides support with a variety of templating language (like EJS, Jade, and Consolidate.js) and an easy integration with the most popular databases (like Cassandra, MySQL, MongoDB). Express.js basically helps manage everything, from routes, to handling requests and views.
Geddy is a full stack open source MVC framework based on ruby on rails methodology with built-in authentication module and websockets integration. Is designed to quickly create real-time web apps and JSON APIs, automating most of the non-trivial tasks such as template rendering, restful router, controllers and models. It also supports a wide range of templates engines, with EJS being the default. Geddy uses Node's built in cluster module which makes use of our multicore processor. Therefore, our node.js application is no longer a single thread application, but leverages multiple cores automatically.

Meteor goes beyond typical JavaScript frameworks. Written in JavaScript on the Node.js platform, Meteor is an open-source Isomorphic Development Ecosystem (IDevE) for efficiently and painlessly developing web and mobile applications. It is efficient by providing a no-fuss API that works the same everywhere and by including many pre-built functionalities most applications need. In addition, it makes the process painless (relatively painless) by removing many of the technical that developers usually have to contend with when developing applications. In short, the Meteor ecosystem seamlessly integrates all the components (tools, libraries, databases, and frameworks [frontend and backend]) necessary for building and deploying applications. The meaning of the terms Isomorphic Development Ecosystem is explained below:

- **Isomorphic.** Isomorphic refers to using the same code on the frontend and the backend, that is, using the same API everywhere (frontend, backend, and even for mobile apps). We can think of isomorphic as homogeneous (of the same kind).

- **Development.** Meteor provides all the tools for the application development life cycle, from setup and configuration tools to API and deployment tools.

- **Ecosystem.** To understand the difference between a JavaScript framework like Backbone.js or Express.js and a development ecosystem like Meteor, consider this: A framework is like an unfurnished kitchen; it has a sink with running water, a stove, and an empty refrigerator; you can cook in such a kitchen of course, but you must bring your own vegetables, pots, containers, and everything else you need, including cabinets for storage. An ecosystem, on the other hand, is like a furnished kitchen with a stocked refrigerator; you need only your desire, maybe a recipe too, and you can cook any kind of meal in the kitchen. The ecosystem provides you with everything you need—all the ingredients, all the utensils, all the containers, all the storage space, and more—to cook complete meals from beginning to end.

**3.3.4 Selected MV* framework**

Finally, Meteor is the selected framework because of it offers developers much more than IDevE, as we will see from the list of features outlined below:

1. Meteor not only has a one-step installation for configuration and setup, but it also has an isomorphic API, which refers to using the same code
on the frontend or backend, or even for mobile and web apps. This saves developers hours since there is no need for developers to wrestle with installing, configuring, and learning disparate libraries, module managers, multifarious APIs, drivers, and the like.

2. It offers not only a frontend framework, like Backbone.js, but also a backend that seamlessly integrates with the frontend, and an easy-to-use API for communicating between the two; this provides developers with straightforward, no-fuss client-server data management (Collections, Models, etc.), server-side methods, and server session management.

3. It provides not only bidirectional persistent communication, but also simplified reactive programming. The reactive programming library works in conjunction with the frontend framework to reactively (that is, instantly and continuously) update the UI whenever dependable data or variables change. Moreover, a Meteor community developer has implemented the Meteor frontend templating engine on the server, providing server side templating for Meteor.

4. It offers not only a stack that includes MongoDB database, but also a front-end representation of MongoDB, called Minimongo, written entirely in JavaScript and available in every connected client. Meteor integrates the two (MongoDB on the backend and Minimongo on the frontend) in a well-conceived manner to mitigate latency, a concept called latency compensation. This result in considerably faster page updates and reloads, leading to a more satisfying user experience for developers and end users alike.

5. It has a standard frontend router (created by a Meteor community member) that implements the best features from other popular frontend routers, and this router also provides server side routing API, even allowing for connect middleware, RESTful endpoints, and the like.

6. Its integrated live browser reload (also known as hot code load and hot code push) not only automatically reloads your live web page whenever you make development changes on the frontend (HTML, CSS, images, JavaScript, etc.), but it also automatically refreshes just the necessary DOM elements on the page (without reloading the entire page), even when there are dependent changes to data on the backend (MongoDB) or frontend (Minimongo).

7. It allows you to use NPM modules and it provides its own build system (a custom package manger) that transcend NPM, providing nearly all the worthwhile and crucial NPM functionalities and more. You can install third party or custom Meteor packages from atmospherejs.com, the official repository for Meteor packages. Therefore, you have access not only to NPM’s 98,000+ modules, but also to Meteor’s AtomophereJS 2600+ smart packages.
8. It has an official testing framework, Velocity. With Velocity, you can use your favorite testing frameworks like Jasmine or Mocha, and run acceptance tests with Selenium.

9. Finally, it not only has a team of dedicated and capable engineers with a vision for changing the world (nearly every team has that), but it also has a lovely bank account: $11.2 million in funding (hardly any team has that). This essentially secures Meteor’s stability for the near future, guarantees frequent updates and timely responses to GitHub issues, and ensures constant interaction with the community.
4 QPides! Implementation on a web platform

This chapter is focused on the implementation of algorithms that solve the table allocation problem automatically in a web environment. The first part contains information about the algorithms, including an explanation of the operation of each of them as well as the methods used to search a valid solution. The second part includes an analysis of the use cases of QPides! necessary to define the technical requirements of the application and track the progress of the project. All source code is on ANNEX B.

4.1 Table allocation algorithms

This subchapter describes the table-allocation algorithms developed during the project. The implementation of the algorithms aims to solve the problem of allocation tables automatically maximizing the occupancy rate and therefore improving not only restaurant’s profitability but also restaurant’s operations. In this implementation, they have been taken into account other factors such as reducing the CPU and memory usage, and minimizing the time it takes to get a valid solution. In this way, it is possible to improve both the efficient use of resources in a restaurant and the use of hardware resources. This allows the algorithms run on a machine with fewer resources and therefore reduce costs, either by creating an own web server or subscribing to a hosting service.

Three algorithms have been implemented: Basic, Random and Backtracking. Basic and Random algorithms are classified as single-table allocation algorithms because their solutions are limited to one table. The backtracking algorithm, however, does not have this restriction since it search the optimal combination of tables, in our case, the combination that maximizes the occupancy rate. That is why the backtracking algorithm is classified as multi-table allocation algorithm. The following briefly define each algorithm:

- **Basic**: Single-table allocation algorithm. The aim of this algorithm is to allocate the first table from a set of available tables whose capacity is greater than or equal to the required capacity. To maximize the occupancy rate, tables list is ordered prior to the allocation, from lowest to highest capacity. It is the simplest algorithm of the three, hence its name. It is also the algorithm running fewer operations. So would a priori the algorithm that has a lower cost of execution time.

- **Random**: Single-table allocation algorithm. It is a modified version of the previous algorithm. Unlike Basic algorithm, instead of allocating the first table of the list, it allocates a table randomly with a uniform distribution. The occupancy rate obtained with this algorithm would be identical to the previous one if all the tables have the same capabilities. However, in restaurants where tables have different capacities, the performance of the algorithm in terms of occupancy rate is impaired, since it not sequentially scans a list of tables ordered from lowest to highest, as the case of the Basic algorithm.
• **Backtracking (BT):** Multi-table allocation algorithm. The aim of this algorithm is to search the optimal combination of tables that maximizes the occupancy rate. To find a solution efficiently the algorithm uses the method of backtracking, hence its name. The theory of this method has been explained in the section 2.4.1. It is the most complex of the three algorithms, since it must search for solutions by combining tables. It is also the algorithm running more operations. Therefore, would a priori the algorithm that has a higher cost of execution time, but also it offers a higher performance in terms of occupancy rate.

The following subsections explain in detail each of the algorithms.

### 4.1.1 Basic
As we have previously explained, the aim of this algorithm is to allocate the first table from a set of available tables whose capacity is greater than or equal to the required capacity. To improve understanding of its operation, it has created a flowchart showing the entire process (see Figure 4-1).

Variables involved:

- **requiredCapacity**: Indicates the number of people for a reservation. This data is included in the request made by the user from the front-end part, for example from a web browser. Its value is limited from the UI and must comprise a number between 1 and the maximum capacity of the restaurant.

- **userId**: It is the unique identifier of the user requesting a reservation and is generated in the initial registration process. The user must be logged into the system to make reservations. So this information also comes from the front-end part.

- **restaurantId**: Before making a reservation is necessary to select a restaurant from the list in the UI. Again, this information is included in the request against the server.

- **availableTables** It is a list of available tables whose capacity is greater than or equal to the value in the `requiredCapacity` variable. This list is updated at the beginning of the reservation process and is ordered from lowest to highest capacity.

- **allocatedTables**: It is a list containing the algorithm solution, i.e., a list of assigned tables. This list is updated at the end of the reservation process. In single-table allocation algorithms, this list only contains one table.

- **reservationId**: It is the unique identifier of the reservation and is generated after the reservation data is stored successful.
The reservation process is as follows:

1. All information about tables is stored in the database, specifically in the `TablesList` collection. A table is available when its last reservation has expired or its status is “Free”. The reservation process starts with getting the set of available tables from the database. Therefore, the algorithm makes a query to `TablesList` collection filtering by `restaurantId`, `capacity`, `reservedUpTo` and `status` fields. The `restaurantId` field must match the ID of the selected restaurant. The `reservedUpTo` field must be less than the current date or the `status` field must be equal to “Free” in order to get only available tables. The function that executes the query is:

   ```javascript
   var getAvailableTables = function(selectedRestaurant){
       return TablesList.find({restaurant_id: selectedRestaurant, $or: [ {reservedUpTo: { $lt: new Date() } }, { status: constants.free} ]}, {sort: {capacity:1}}).fetch();
   }
   ```

   This function returns a list of potentially available tables for a reservation, ordered by capacity from lowest to highest.
2. The first table whose capacity is greater than or equal to the required for reservation represents the solution of this algorithm. The content of the new reservation is inserted into the database and allocated table fields are updated. The reservedUpto is set to a new reservation deadline, typically thirty minutes added to the reservation date, and the status field is set to “Occupied”.

4.1.2 Random
This algorithm is a modified version of the previous algorithm and therefore its behavior is almost identical. The only difference is that in step two, instead of allocating the first table in the availableTable list that meets the reservation requirement, it allocates a table randomly with a uniform distribution that represents the solution of this algorithm. The rest is the same.

4.1.3 Backtracking
It is the only multi-table allocation algorithm implemented in this thesis. The aim of this algorithm is to find the optimal combination of tables that maximizes the occupancy rate.

Explore all possible combinations of tables with a simple method, such as Generate and Test, would perform \(2^N\) allocations, where N is the number of available tables. The calculation of the number of allocations is an exponential in base two since the problem has been modeled as a binary CSP. That is, the domain of the variables can only take values of zero or one. Zero means no assigned table and one means assigned table. For example, in a restaurant with 20 available tables, the number of allocations would amount to 1,048,576 \((2^{20})\).

Searching with BT can be much more efficient than generate and test. Generate and test is equivalent to not checking constraints until reaching the leaves. Checking constraints higher in the tree can prune large subtrees that do not have to be searched, as it is explained in the section 2.4.1. Therefore, BT is the method used to implement the algorithm.

The BT method does not follow rules for finding the solution because is simply based on a systematic search. This means that it should try everything possible to find the solution or find that there is no solution. To achieve this purpose, the search is divided into partial searches or subtasks. In addition, these subtasks typically include more subtasks, so the overall treatment of these algorithms is recursion.

This method is called Backtracking because in the case of not finding a solution in a subtask, it goes back to the original subtask and it proves another thing (a new subtask different to those previously tested).

Its operation is similar to go in depth a graph, where each subtask is a node in the graph. The point is that the graph is not defined explicitly (as a list or matrix), but implicitly, that is, that will be created as all branches of the graph are explored. This graph is usually a tree, or does not contain cycles. This
means that when looking for a solution is generally impossible to reach the same solution \( X \) from two different subtasks \( A \) and \( B \). The graphical explanation of this is shown in Figure 4-2.

![Figure 4-2 Procedure of BT method for finding solutions](image)

It often happens that the tree or graph generated is so big to find a solution or find the best solution among many possible is computationally very expensive. In these cases usually applies a series of restrictions, so that some of the branches can be pruned, i.e. not do certain subtasks. This is possible if at some point of the search it can be shown that the solution obtained from that point will not be better than the best solution obtained so far. If it is done properly, pruning does not prevent find the best solution.

Sometimes it is impossible to prove that doing pruning is not hiding a good solution. However, the problem may not ask for the best solution, but one that is reasonably good and whose computational cost is quite small. That's a good reason to increase the restrictions when a node is explored. Perhaps the best solution is lost, but an acceptable solution will be found in a short time.

For the design of our backtracking algorithm, the following elements were taken into account:

a) Representation of the problem as a search tree.

b) Representation of the solution in an ordered list \( (X_1, X_2, ..., X_n) \).

c) A function to determine whether the solution found is a valid solution.

d) A function to prune parts of the search tree.

```
procedure test (step: stepType)
    for each candidate do
        select candidate
        if acceptable then
            begin
            record_candidate
            if incomplete_solution
            test (next_step)
            else
            store_solution
            delete_candidate
            end
        until out_of_candidates
    end
end procedure
```

Code 4-1 Generic structure of backtracking algorithms [17]

The backtracking algorithm implementation includes new variables: `curSolution`, `stage`, `finalSolution`, `finalCapacity` and `finalBenefit`, in addition to the variables of the Basic and Random algorithms described above. The meaning of these variables is:

- **`curSolution`**: Integer array that stores the current allocated tables. This array dynamically evolves until all possible combinations of the graph have been contemplated.

- **`stage`**: It indicates the current level within the binary search tree. Its initial value is 0 because the depth-first search begins at the root node.

- **`finalSolution`**: Integer array that stores the final allocated tables.

- **`finalCapacity`**: It indicates the total capacity of the allocated tables.

- **`finalBenefit`**: It indicates the occupancy rate of the allocated tables. It is the ratio of `requiredCapacity` to `finalCapacity`.

- **i**: Variable indicating whether or not the current table (node) is selected in the tree. 0 means not selected and 1 means selected.

The flowchart of the algorithm for allocating tables using the BT method is shown in Figure 4-3.
In order to reduce the computational cost when exploring all possible solutions, pruning function applies the following constraints:

1. The total capacity of the allocated tables in the current solution should be greater than the reservation capacity required by the client.

2. The benefit of the current solution is worse than the benefit of the best solution found so far.

To make pruning both constraints must be met. If the benefit of the current solution is worse than the benefit of the best solution found until that point, add another table to the current solution can never provide a greater benefit, given that the list of available tables is sorted from lowest to highest capacity. This pruning does not prevent finding the best solution in addition to improving the performance of the algorithm.
4.1.4 MongoDB Limits
Most popular relational databases today support “ACID” properties – Atomicity, Consistency, Isolation and Durability. Developers and DBA’s working with relational databases have a good understanding of ACID behavior. However, when working with NoSQL databases like MongoDB there are some key differences that are important to understand. MongoDB offers great flexibility in storage, schema and scaling, but relaxes some of the ACID properties [18]. It is important to understand the limitations of MongoDB to solve some of the problems that have appeared during the implementation of QPides!

4.1.4.1 Atomicity
In an atomic transaction, a series of database operations either all occur, or nothing occurs. A guarantee of atomicity prevents updates to the database occurring only partially, which can cause greater problems than rejecting the whole series outright. In other words, atomicity means indivisibility and irreducibility [19].

MongoDB write operations are atomic only at the level of a single document. If you are modifying multiple subdocuments inside a document, the operation is still atomic. If you are modifying multiple documents, the operation is not atomic. Then, to get an atomic behavior across multiple documents is necessary to use a “Two phase commit” pattern. There is great example from the MongoDB documentation on how to implement this pattern [20].

Although finally an atomic behavior to implement QPides! was not necessary, this is an important limitation that has wanted to highlight.

4.1.4.2 Isolation
In database systems, isolation is a property that defines how/when the changes made by one operation become visible to other concurrent operations [21]. There are multiple ways to achieve Isolation with MongoDB operations. Here are some:

1. The “findAndModifyOperation()” is one of the simplest ways to query and modify existing documents. The command can return either the previous values of the documents or the new updated values of the documents. You can also sort the matching documents, upsert and select which fields need to be returned.

2. “Update if current” pattern. This pattern is specified in the MongoDB documentation [20]. It involves more manual work but gives you more control.

3. $isolation operator. The $isolation operator provides a way to isolate writes to multiple documents. How the $isolation operator does not provide all or nothing guarantee, it is necessary to use some of the atomicity techniques specified in the first section to achieve that.
During the implementation of QPides! appeared a problem related to isolation. Until that time, there was no system able to control whether any of the assigned tables had been previously reserved by another concurrent user. Consequently, this resulted that sometimes the same tables was assigned to different reservations for the same time slot. This violated an explicit limitation of our system as a table can accommodate many people as its capacity, but no more.

To solve it, the *Update if current* approach is applied together with the *$isolation* operator. The *counter* field was added to the documents in the *TablesList* collection. This field indicates the number of times that a table has been modified and is taken into account in the process of updating the document. If the value of the *counter* field of the allocated table, obtained at the start of the reservation process, does not match with the same field of the current document in the collection, MongoDB does not alter the document. If it matches, the document is modified increasing the *counter* field. Only if none of the allocated tables have been modified a new document is inserted in the *ReservationList* collection with the reservation data.

### 4.1.4.3 Concurrency

MongoDB uses locks to prevent multiple clients from updating the same piece of data at the same time. MongoDB 2.2+ uses “database” level locks. So when one write operation locks the database all other write operations to the same database (even if they are to a separate collection) are blocked waiting on the lock. MongoDB uses reader-writer locks that allow concurrent readers shared access to a resource, such as a database or collection, but give exclusive access to a single write operation.

### 4.2 Use case analysis

The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements. Mistakes made in the definition of the requirements are the most dangerous, since its consequences affect all other stages of the life cycle of the software system. In order to fix this situation as far as possible, there is the use case analysis. It is a technique used extensively in the analysis phase for gathering usage requirements of a new software program. It helps the developer to design a system from the user viewpoint. The primary goals of a use case analysis are designing a system from the user’s perspective, communicating system behavior in the user’s terms and specifying all externally visible behaviors. Through the use case diagram, it illustrates the activities that are performed by a user (known in Unified Modeling Language (UML) as an "actor") and the system’s responses.

The use case analysis has been used to manage the project requirements of QPides! application. First, we define a list of possible actors that interact with our application:
- Consumer
- Restaurant Manager
- Administrators
- Chef
- Hostess

Then, we create one use case diagram for each actor.

![Figure 4-4 Chef use case (left) and Restaurant Manager use case (right)](image)

- **Chef**: manage the menu of the restaurant. It means add, delete, or edit the menu, including pictures, description, ingredients and price. He can see pending items to cook as a list, organized by descending date, i.e., the oldest order is the first in the list. In order to serve faster, items are assigned to specific waiters. Also, the chef has the possibility to send delivery time estimation to consumers, so they know roughly how long to wait before receiving an order. All use cases of this actor are shown in Figure 4-4.

- **Restaurant manager**: he can manage employee accounts. For example, to register employees in the system, modify employee data, or enable / disable the account depending on whether the employee works or not now. He has the privileges to publish information about the restaurant, for example offers, promotions, respond to consumer messages. He can also generate reports based on expenses, income, resources, higher occupancy hours, best-selling dish and most frequent consumers. Furthermore, he can see the hours of each employee as the login system keeps track in and out of the QPides! application. All use cases of this actor are shown in Figure 4-4.
Hostess: have permissions to view information related to the status of the restaurant tables in real time, for example, if a table is occupied or not, or if cleaning table is pending. It can also assign tables manually if the allocation algorithm is not running. The use cases of this actor are shown in Figure 4-5.

Administrators: They are responsible for maintenance of the application and therefore they have the highest privileges. One of its functions is to register restaurant managers who want to use the application in their facilities. The use case of this actor is shown in Figure 4-5.

Consumer: is the actor who most interact with the application. Therefore we have grouped the different use cases by themes (see Figure 4-6):

- Social network: has the option to rate the service through a system of 0 to 5 points, recommend restaurants to friends and see reviews from other consumers.

- Favorites: He can manage your favorite restaurants adding and removing restaurants from the list of favorites.

- Authentication: He can register itself in the system as well as getting in and out of the system using a combination of username and password.
Figure 4-6 Consumer use case

- **Restaurants**: He can see detailed information about the restaurant such as type of cuisine, availability, and contact information. He can also look for restaurants that are closest to their location using geolocation.

- **Ordering**: He can manage your orders, such as adding items to the shopping cart or see the history of your orders. It also has a direct communication channel with the restaurant.

After producing the initial visual list of use case actors and goals, we create an initial use case grid, which provides the basis for the use case index (see Table 4-1). This use case index will serve as a master inventory to help write effective
use cases for the requirements phase of the project. Although this thesis has focused only on the implementation of table allocation algorithms on a web environment, as part of the QPides! application, it has done an analysis of all use cases of the application.

There are use cases that are used by more than one actor. To simplify the list of use cases and eliminate those that are duplicates, have been established user groups that extend functionality to specific actors. These groups are:

- Generic users: All actors belong to this group.
- Restaurant Staff: This group includes waiters, hostess and chefs.

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>Use Case Name</th>
<th>Primary Actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sign up</td>
<td>Consumer</td>
</tr>
<tr>
<td>2</td>
<td>Sign in</td>
<td>Generic user</td>
</tr>
<tr>
<td>3</td>
<td>View orders</td>
<td>Consumer</td>
</tr>
<tr>
<td>4</td>
<td>Edit account</td>
<td>Consumer</td>
</tr>
<tr>
<td>5</td>
<td>Reset password</td>
<td>Generic user</td>
</tr>
<tr>
<td>6</td>
<td>Sign out</td>
<td>Generic user</td>
</tr>
<tr>
<td>7</td>
<td>View geo-localized deals</td>
<td>Consumer</td>
</tr>
<tr>
<td>8</td>
<td>Find restaurant</td>
<td>Consumer</td>
</tr>
<tr>
<td>9</td>
<td>View restaurant info</td>
<td>Consumer</td>
</tr>
<tr>
<td>10</td>
<td>View deals from Favorites</td>
<td>Consumer</td>
</tr>
<tr>
<td>11</td>
<td>Add to Favorites</td>
<td>Consumer</td>
</tr>
<tr>
<td>12</td>
<td>Remove from Favorites</td>
<td>Consumer</td>
</tr>
<tr>
<td>13</td>
<td>Book online</td>
<td>Consumer</td>
</tr>
<tr>
<td>14</td>
<td>Check-in to restaurant</td>
<td>Consumer</td>
</tr>
<tr>
<td>15</td>
<td>Check-in to table</td>
<td>Consumer</td>
</tr>
<tr>
<td>16</td>
<td>Check-out from restaurant</td>
<td>Consumer</td>
</tr>
<tr>
<td>17</td>
<td>Check-out from table</td>
<td>Consumer</td>
</tr>
<tr>
<td>18</td>
<td>View menu</td>
<td>Consumer</td>
</tr>
<tr>
<td>19</td>
<td>Call the waiter</td>
<td>Consumer</td>
</tr>
<tr>
<td>20</td>
<td>Add items to shopping cart</td>
<td>Consumer</td>
</tr>
<tr>
<td>21</td>
<td>Manage the shopping cart</td>
<td>Consumer</td>
</tr>
<tr>
<td>22</td>
<td>Place the order</td>
<td>Consumer</td>
</tr>
<tr>
<td>23</td>
<td>Redeem a coupon code</td>
<td>Consumer</td>
</tr>
<tr>
<td>24</td>
<td>Pay the order</td>
<td>Consumer</td>
</tr>
<tr>
<td>25</td>
<td>Rate the restaurant</td>
<td>Consumer</td>
</tr>
<tr>
<td>26</td>
<td>Manage friends</td>
<td>Consumer</td>
</tr>
<tr>
<td>27</td>
<td>Recommend restaurants</td>
<td>Consumer</td>
</tr>
<tr>
<td>28</td>
<td>View the occupation on the floor layout.</td>
<td>Hostess</td>
</tr>
<tr>
<td>29</td>
<td>Allocate tables.</td>
<td>Hostess</td>
</tr>
<tr>
<td>30</td>
<td>Send Timesheet</td>
<td>Restaurant Staff</td>
</tr>
<tr>
<td>31</td>
<td>See items to serve</td>
<td>Waiter</td>
</tr>
<tr>
<td>Use Case Element</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>Use Case Number</td>
<td>ID to represent your use case</td>
<td></td>
</tr>
<tr>
<td>Application</td>
<td>What system or application does this pertain to</td>
<td></td>
</tr>
<tr>
<td>Use Case Name</td>
<td>The name of your use case, keep it short and sweet</td>
<td></td>
</tr>
<tr>
<td>Use Case Description</td>
<td>Elaborate more on the name, in paragraph form.</td>
<td></td>
</tr>
<tr>
<td>Primary Actor</td>
<td>Who is the main actor that this use case represents</td>
<td></td>
</tr>
<tr>
<td>Precondition</td>
<td>What preconditions must be met before this use case can start</td>
<td></td>
</tr>
<tr>
<td>Trigger</td>
<td>What event triggers this use case</td>
<td></td>
</tr>
<tr>
<td>Basic Flow</td>
<td>The basic flow should be the events of the use case when everything is perfect; there are no errors, no exceptions. This is the &quot;happy day scenario&quot;. The exceptions will be handled in the &quot;Alternate Flows&quot; section.</td>
<td></td>
</tr>
<tr>
<td>Alternate Flows</td>
<td>The most significant alternatives and exceptions</td>
<td></td>
</tr>
</tbody>
</table>

Table 4-1 Use case grid

Use cases have been analyzed in detail one by one according to the Table 4-2. The analysis of the use cases is included in the Annex.
5 Experimental measurements

5.1 Introduction
It has conducted a series of load tests against the server to evaluate the performance of each of the algorithms described in section 4.1.3. and to evaluate their performance in terms of response time per booking (measured in seconds) and occupancy rate per booking, to assess its position with respect to the simple and random algorithms as a candidate for application in restaurants.

5.2 Defining the test scenario
Tests have been conducted on an environment consisting of two computers connected by a local area network (LAN). A computer functions as the application server and the other as a client.

The server includes a tool called Kadira.io that allows monitoring parameters such as memory usage and CPU (see Figure 5-1). This system information is necessary to check the current state of the app and debug any issues.

![Figure 5-1 Kadira.io performance monitoring system](image)

The client uses MeteorDown, a tool for load testing against servers developed with Meteor. This tool allows us to customize the load test using the JavaScript language and uses the DDP protocol to communicate with the Meteor application.

The main features of the PC used in the tests are:

**Server**
- Operating System: Linux Mint 17.1
- System Manufacturer: ASUSTeK Computer Inc.
- System Model: K53SJ
- Processor: Intel(R) Core(TM) i5-2410M CPU @ 2.30GHz (4 CPUs)
- Memory: 4096MB RAM
5.3 Load testing

Any load-testing project should start with the development of a model for user workload that an application receives. This should take into consideration various performance aspects of the application and the infrastructure that a given workload will impact. A workload profile is a key component of such a model. Choosing the workload profiles representative of anticipated real load over time (whether it is an everyday usage scenario or a high peak) results in more accurate answers to the “main questions of load testing” such as, “Will my site support N users performing a search at the same time?” and “What is the highest number of users that my site will support?”.

The workload model then attempts to approximate real life usage scenario. To determine the workload of QPides!, it is considered the set of possible actions that a user can perform. In our case, for a typical reservation system for restaurants, is taken into account the following:

- Connect to the home page.
- Log on to the application.
- Select a restaurant.
- Request a reservation.
- Log out from the application.

Given the above, and given the large number of degrees of freedom available, the user model applied in our load tests is as follows:

1. The user connects to the server.
2. The user waits a random time that follows a normal distribution $N (\mu, \sigma)$ with five seconds of average and two seconds of standard deviation.
3. The user selects a restaurant randomly from the collection.
4. The user selects a random capacity between one and ten.
5. The user makes a reservation request specifying one of the implemented table allocation algorithms.
6. The user waits a random time that follows a normal distribution $N (\mu, \sigma)$ with five seconds of average and two seconds of standard deviation.
7. The user disconnects from the server.

Additional data on the load tests:
• Reservations expire instantly. This means that all tables will be available for allocation.
• The total number of restaurants is five.
• Each restaurant has a total of twenty tables distributed in ten tables for two, eight tables for four and two tables for eight.
• Each algorithm has been tested with different levels of concurrency: 1, 25, 50, 100, 200, 400, 800 users.
• The duration of each test is determined by the time it takes the system to manage C*10 reservations, where C is the level of concurrency.

5.4 Results:

5.4.1 Average time of reservation acceptance

The average time of reservation acceptance is shown in the graph Figure 5-2. It is defined as the time between the client requests a reservation and receive an acceptance. The Y-axis represents the response time on a logarithmic scale (base 10). The X-axis represents the number of concurrent users on the server.

The backV1 version stands out from the rest of the algorithms for its longer response time. As expected, this is due to the larger number of operations that executes when searching the best combination from among all possible. It is the only one that is not able to serve 400 concurrent requests in our test scenario because of its high computational cost.
The results of the other algorithms show no significant difference between them in this test. However, it is important to highlight the performance of the backV2 and backV3 algorithms, since they are positioned close to single-table allocation algorithms: basic and random. This means that pruning and search for the first acceptable solution provide a shorter response time than the backV1 version of backtracking, thus reducing its computational cost. We will see later if these algorithms also provide good performance in terms of occupancy rate compared to the backV1 version.

The response time, in all scenarios, increases as it increases the level of concurrency. This is due to hardware limitations and involves a degradation of the quality of the web service.

5.4.2 Reservations denied

The Figure 5-3 shows the performance of algorithms in terms of number of denied reservations. The Y-axis represents the number of denied reservations on a logarithmic scale (base 10). The X-axis represents the number of concurrent users on the server. The reservation is denied if any of the following conditions is met:

1) There are no tables available.
2) There is no combination of tables that meets the necessary requirements.
3) Some assigned table has been modified by another concurrent user during the reservation process.

Figure 5-3 Reservations denied
As explained above, the reservations expire instantly according to the configuration of our tests. This means that all tables are always available for allocation, and therefore, the algorithms can never deny a reservation because of the first reason. Neither is it possible to reproduce the second case, since the capacity required by consumers is always less than or equal to the maximum capacity of the existing tables. So, with this configuration it is only possible to deny a booking for the third reason, when some assigned table has been modified by another concurrent user during the reservation process.

Again, the basic backtracking algorithm is showing the worst performance, especially in environments with a high level of concurrency (> 100). Due to its low performance it is not possible to serve 400 concurrent requests, like the rest of algorithms. The results obtained in the previous section are related to these, as the probability of rejecting a reservation is accentuated by various reasons:

1. The response time is greater, then it is more probable than another concurrent user has reserved the same table/s.

2. Assigning multiple tables in the same booking increases the probability of rejection since only a reservation is accepted if none of the assigned tables has been modified by another concurrent user during the reservation process.

Thus, the random algorithm is rejecting fewer reservations. The probability of assigning a reserved table to another user is lower for several reasons:

1. Low response time. This is due in turn to its low computational cost since it does not combine tables to find a valid solution.

2. Random component. Unlike the basic algorithm, the random component reduces the chance of reserving the same table for the same required capacity.

The backV2 and backV3 versions have a reasonable performance, since they are located very close to the simplest algorithm, considering that they attempt to maximize the occupancy rate.

### 5.4.3 Occupancy rate

The occupancy rate in reservation systems is the ratio between the required capacity and the reserved capacity. Generally, the occupancy rate is measured in percentage. The results obtained in terms of occupancy rate for each algorithm are shown in the Figure 5-4.
Experimental measurements

Unlike previous results, single-table allocation algorithms are those that offer poorer performance. The random algorithm is in the last position with an occupancy rate of 58.89%. This is an algorithm that does not maximize occupancy since allocates a random table whose capacity is greater than the required capacity. The basic algorithm is in fourth position with an occupancy rate of 69.69%, but still far from backtracking algorithms. This algorithm is the most optimal for single-table allocations.

Backtracking algorithms provide the best performance. It was expected, since they are designed to maximize the occupancy rate. It is important that all versions of backtracking reach an occupancy rate of around 90%. This means that pruning is done correctly in the backV2 algorithm because it does not prevent looking for the best solution and the constraints implemented in the backV3 algorithm are good enough.

5.4.4 Resources consumption

This section shows the results about the use of server resources, such as memory usage and CPU.

5.4.4.1 RAM

The backtracking algorithm must store the state variables of the search tree from the root node to the current node. The implementation of the algorithm backV2 and backV3 means saving memory space when compared to the backV1 version. The random and basic algorithms use less memory because they look a solution of one table and therefore should not analyze any tree. The results obtained are shown in Figure 5-5.
5.4.4.2 CPU

CPU time is the amount of time for which a central processing unit (CPU) was used for processing instructions of a computer program. Figure 5-6 presents the CPU usage of QPides! server.

The \textit{backV1} version stands out from the rest of the algorithms for its higher CPU consumption. As expected, this is due to the larger number of operations that executes when searching the best combination from among all possible its computational cost.
Figure 5-6 CPU usage
6 Conclusions

Design, study and implementation of new systems of allocation tables are necessary in order to improve the performance and efficiency of the restaurants, both in terms of quality of service and in terms of profitability. This process of evolution is required to assume the relentless popularization of web services and in particular, emerging online reservation systems that pose new challenges with payment gateways, description of products and services and information about the service from start to finish.

This thesis considers new table-allocation algorithms that contribute to the search for new mechanisms of online restaurant reservation in order to improve efficiency and contribute to the evolution of web services. These algorithms solve a constraint satisfaction problem, looking the best combination of tables at any given time that maximize the occupancy rate. By creating a system of load testing it has been possible to study the behavior of the algorithms under different level of concurrency.

The solution is based on constraint programming over a web environment. It has been programmed using Meteor, a Javascript modern web framework. For the design of the algorithms, some elements were taken into account: representation of the problem as a search tree, representation of the solution in an ordered list, a function to determine whether the solution found is a valid solution, a function to prune parts of the search tree. The results obtained in the experimental measurements prove that searching with BT is much more efficient. Performing a properly pruning does not prevent find the best solution and reduces the computational cost. The web application also manages a MongoDB database with dummy data about restaurants, a complete user account (login, logout, account creation, email validation, recovery password, roles), URL routing, sessions and geolocation.

The initial goals of this work were to design tables allocation algorithms and evaluate their performance by implementing the restaurants reservation system (QPides) in a web environment. It can be concluded that the work has successfully managed to meet the goals as some designed table allocation algorithms maximizes the restaurants occupancy rate with negligible hardware performance degradation, making it ideal for restaurants with a high occupation and with strict requirements of efficiency in use of resources.

6.1 Environmental study

It might seem that this project does not have a significant impact on the environment. However, it does have an effect on energy saving, which is an important factor in environmental sustainability.

Thanks to QPides! the user has real-time information of restaurants, plus instant confirmation of booking if there is availability. Thus, many unnecessary travels in vehicles that consume fossil fuels are avoided, reducing environmental pollution.
7 Future research

During the analysis of the data for this Thesis, I realized that there are additional ways to try to improve the restaurant management that may be interesting to further analyze but they were outside of the scope of this Thesis. These ways are described below:

- The customer must be persuaded to accept another time if a request cannot be accommodated on the current booking sheet.

- Estimate the expected duration of the meal based on the characteristics of the booking (including time, day of the week, and party size).

- Manage 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.

- Develop the full solution:
  - Geo-localized advertisement for possible consumers
  - Targeted customer offers
  - Manage menu/catalog of the services/products offered
  - Control product availability
  - Smart and secure order payment management
  - Business data mining

- Estimate about future table demand, i.e. about the distribution of future requests by number, size, and booking time, to build up a seating plan that maximizes the table usage.

- Maintain a list of reserves as a contingency plan in case of cancellations or no-shows. For instance, customers whose requests have been initially rejected but that are willing to be contacted in case a table should become available.
8 Bibliography


