# Experimental analysis of optimization techniques on the road passenger transportation problem.

Beatriz López,Victor Muñoz, Javier Murillo *

*University of Girona, Campus Montilivi, edifice P4, 17071 Girona, Spain.*

Federico Barber, Miguel A. Salido, Montserrat Abril, Mariamar Cervantes

*DSIC, Universidad Politécnica de Valencia, Spain*

Luis Fernando Caro, Mateu Villaret

*University of Girona, Campus Montilivi, edifice P4, 17071 Girona, Spain.*

**Abstract**

Analyzing the state of the art in order to tackle a new problem is always a mandatory issue. Literature offers surveys based on summaries of previous works, often based on theoretical descriptions of the methods. An engineer however, requires some evidence from experimental evaluations in order to make the appropriate decision when selecting a technique for a problem. This is what we have done in this paper: experimentally analyze a set of representative techniques of the state of the art in the problem we are dealing with, namely the road passenger transportation problem. This is an optimization problem in which drivers should be assigned to transport services fulfilling some constraints and minimizing some function cost. The experimental results have provided us with a good knowledge of several properties of the methods, as modeling expressiveness, anytime behavior, computational time, memory requirements, parameters, and free downloadable tools. From our experience, we are able to choose a technique to deploy our problem. We hope that this analysis can be also helpful for other engineers facing a similar problem.

*Key words:* Optimization, constraints, search, distributed problems, metaheuristics, bioinspired approaches

* Corresponding author: Beatriz Lopez (beatriz.lopez@udg.edu)

## 1 Introduction

Whenever either an engineer or researcher faces a new problem, he/she needs to review the state of the art related to it, in order to be sure that the problem has not already been solved in the past, and to analyze what the most promising approach to follow could be in order to achieve the appropriate development. In each discipline, surveys are being published periodically. However, most of the surveys are summaries of previous works which give some hints on what the techniques are. But from the information provided in the surveys it is often difficult to evaluate the suitability of the techniques for the problem at hand, unless the techniques are being applied to the particular problem. This has been our situation.

We are dealing with the *road passenger transportation problem*. Road passenger transportation has for years been a matter of concern for traffic authorities in order to minimize bus accidents. Regarding buses, European law is also evolving in order to control professional driving licences and driving times, with the aim of assuring the maximum guarantees to the citizens that use road passenger transport. These new laws and regulations are placing a considerable number of requirements to inter-urban transport companies related to just-in-time services. That is, services required within a short period of time, usually from one day to the next (conference events, holidays, excursions). The problem of these companies is to allocate once a day drivers to required services.

The road passenger transportation problem clearly fits in the category of optimization problems, in which resources (drivers) should be assigned to tasks (transport services) fulfilling some constraints and minimizing some function cost. Even that in most cases these problems are solved by the state of the art techniques, there are still a lot of recent papers dealing with the driver allocation problem, as (Abbink et al., 2007), (Ramalhinho Lourenço et al., 2006), and (Laplagne et al., 2005), meaning that the problem is still open due to the different problem specificities the engineers should tackle. There appears to be no general guidelines to choose the appropriate technique given the specific description of a problem. In our case, services are inter-urban, just-in-time scheduling is required, and new legislation define new constraints. In addition, computational efficiency is the main feature with which researchers use to compare techniques, while other features are also important. For example, expressiveness can be an interesting issue regarding the solution interpretation or even the posterior incorporation of robustness into the solution. Our intention with this experimental analysis is to contribute in the understanding of the current state of the art techniques beyond a pure efficiency analysis, but with other interesting features as model expressiveness, anytime behavior, memory requirements, parameter tuning

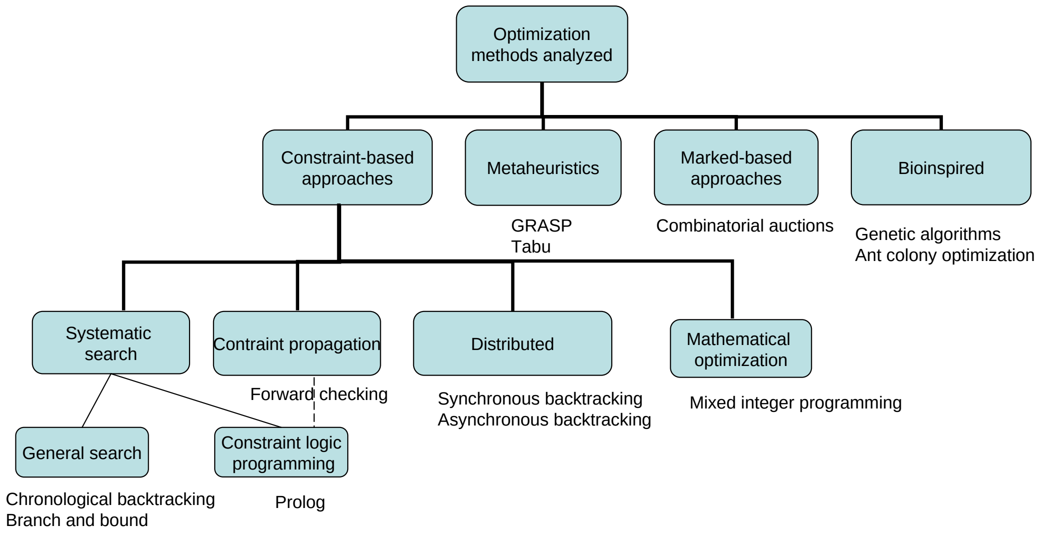

Fig. 1. Methods analyzed grouped in categories.

and tools availability. The characterization of the techniques provided in this paper could be a first step towards the selection of the appropriate general technique when dealing with similar resource allocation problems and looking for particular techniques properties.

We have experimentally analyzed a total of 12 techniques, grouped in four categories: bioinspired methods, metaheuristics, constraint-based methods, and market-based methods (see Figure 1). Note that this classification is not crisp, since some of the methods can be assigned to more than one category. For example, genetic algorithms can be classified as metaheuristics or bioinspired methods. Nevertheless, we think that the methods chosen to perform the analysis widely covers the different methods available in the literature in both Operational Research and Artificial Intelligence fields, including the newest distributed and market-based approaches. It is important to note that each technique requires a specific approach to the problem. So for each technique analyzed we provide some generalities regarding the technique, the modeling of the problem according to its requirement, and the experimental results obtained.

By providing an illustrative use of several techniques for dealing with the same combinatorial problem we intent to help both, beginners in the optimization world and experts that wish to update their scope on the area.

## 2 Problem description

In the road passenger transportation problem we are presented with a set of resources (drivers) $D = \{d_1, \ldots, d_n\}$ and a set of tasks (services)

$S = \{s_1, \ldots, s_m\}$ to be performed by using the resources. The problem consists of finding the *best* assignment of drivers to services, given a cost function and subject to the constraints and preferences provided by the administration (local, national or European). We are dealing, then, with a *constraint optimization problem*, and particularly, with a scheduling problem, since we are interested in knowing the scheduling of each driver in order to deploy all the requested services. More concretely, since drivers are our resources, we are dealing with a resource allocation problem.

There is an alternative approach to the problem, in which services are grouped according to possible driver journeys. A **journey** is then defined as the set of services that can be assigned to a single driver (journey duties driver). Journey generation is known as the crew scheduling problem, that is complemented by the rostering problem in which journeys are assigned to drivers (Ramalhinho Lourenço et al., 2001). In our experimental study, tackling the problem as a whole (**service** approach) or in two steps (journey approach) depends on the modeling capacities of the techniques.

Regarding the problem complexity, when only a driver is assigned to a journey, as in our case, the crew scheduling problem is known to be an instance of the set partitioning problem (Laplagne et al., 2005; Kohl., 2003) that is NP-complete (Saxena, 2007; Garey and Johnson, 1979)[1]. Similarly, when facing the problem according to the service approach, the complexity is also NP-complete. As far as we are looking for solutions that minimize the allocation costs, we are dealing with a NP-hard problem.

## *2.1 Problem formalization*

**Definition 1** *A* service *is a tuple $s_i =< sl_i, fl_i, st_i, ft_i >$ where $sl_i$ is the start location, $fl_i$ the final location, $st_i$ the initial time, and $ft_i$ the final time ($st_i < ft_i$).*

**Definition 2** *A* driver *is a tuple $d_i =< bc_i, kmc_i, sl_i, fl_i, hw_i, hc_i >$ where $bc_i$ is the basic cost, $kmc_i$ is the cost per kilometer, $sl_i$ is the start location, $fl_i$ is the final location (often $sl_i = fl_i$), $hw_i$ are the accumulated two week hours, and $hc_i$ is the cost per time unit.*

There are two kinds of services to be considered: requested and intervening. Requested services (or services for short) are the ones that customers have applied for, while intervening services are those required to move the driver

[1] When more than one driver can be assigned to a journey (for reserve situations, for example (Ramalhinho Lourenço et al., 2001)), the problem is known to be an instance of the set covering problem, also NP-complete.

from the end location of a service to the start location of the next service assigned to him.

**Definition 3** *Given two services, $s_i$ and $s_j$, with $ft_i < st_j$, an* intervening service *between $s_i$ and $s_j$ is defined as a tuple $s_{i-j} =< sl_{i-j}, fl_{i-j}, st_{i-j}, ft_{i-j} >$ where $sl_{i-j}$ is the start location (with $sl_{i-j} = fl_i$), $fl_{i-j}$ the final location (with $fl_{i-j} = sl_j$), $st_{i-j}$ the initial time, and $ft_{i-j}$ the final time, with $st_{i-j} > ft_i$ and $ft_{i-j} < st_j$.*

Given a set of services $S$, and a set of drivers $D$, a total number of $k$ intervening services could be required. Let $I$ be the set of such intervening services. Then,

**Definition 4** *An* allocation based on services *is a list of pairs $A_i = [(s_1, d_{i_1}), (s_2, d_{i_2}), ...(s_l, d_{i_l})]$ where $s_i \in S \cup I$, $d_j \in D$, and in which all constraints are satisfied. Furthermore, $\bigcup_{s_i \in (A_i \setminus I)} = S$, that is, all requested services are covered, and $\bigcap_{s_i \in (A_i \setminus I)} = \emptyset$, that is, no service is repeated.*

Among all the possible constraints of the problem (see (López, 2005) for a complete description of the problem) the following constraints have been considered for the experimental study:

- Overlapping: A driver cannot be assigned to two different services with overlapping times. In addition, a driver assigned to a service that ends at time $t$ and location $l$ cannot be assigned to another service that starts at time $t + 1$, unless the location of the new service is the same ($l$).
- Maximum driving time (MaxDT): the driving time required for both the requested and intervening services.
- Maximum journey length (MJ): the addition of the driving time plus the free time among assigned services cannot be over the maximum journey length allowed.
- Maximum driving time per two-weeks (MTB): the maximum driving time per two weeks cannot be over 90 hours.

**Definition 5** *A* journey *is an ordered set $j_i = \{s_1, ..., s_p\}$ where $s_j \in S \cup I$, in which the overlapping, maximum driving time and maximum journey length constraints are satisfied.*

**Definition 6** *An* allocation based on journeys *is a list of pairs $A_i = [(j_1, d_{i_1}), (j_2, d_{i_2}), ...(j_l, d_{i_l})]$ where $j_k$ is a journey, $d_k \in D$, $S \subset \bigcup_k j_k$ (all services are covered) and in which all constraints are satisfied.*

The *cost function* that measures the individual cost of a driver $i$ in an allocation $A_k$ is the following:

$$cost(A_k, d_i) = bc_i + \frac{(distance(A_k, d_i) * kmc_i)}{\alpha} + (h(A_k, d_i) * hc_i)\beta \qquad (1)$$

where $distance(A_k, d_i)$ is the distance covered by the driver in the $A_k$ allocation measured in kilometers, $h(A_k, d_i)$ is the journey of the driver in the $A_k$ allocation (including non occupied time) and $\alpha$ and $\beta$ are parameters of the cost function whose purpose it to make kilometers and hours, which have different scales (kilometers are usually defined in [0,100] while hours in [0,24]), comparable. After several tries, we have set $\alpha = 10.0$ and $\beta = 7.0$ respectively.

The *cost function* that measures the cost of an allocation $A_k$ is defined as the addition of the individual costs of the drivers $cost(A_k, d_i)$, that is,

$$C(A_k) = \sum_{i \in \{1,\dots,n\}} cost(A_k, d_i). \tag{2}$$

The road passenger transportation problem consists of finding the allocation that minimizes the cost ( $argmin_{\forall i}(C(A_i))$ ) subject to the above constraints.

## *2.2 The workbench*

In order to experimentally analyze the different techniques, up to 70 problem instances have been generated with different complexity. The data corresponding to services (start and end destinations, and start and end times) and drivers (basic cost, cost per kilometer, cost per time unit, starting and ending location, and cumulated driving hours) have been generated randomly for each example. Then, the first instance has been defined with the first generated service and driver; the second instance with the second two generated services and drivers; and so on until the 70th example, being the complexity of the 70th instance greater than in a real case of the application we are dealing with. In this sense, we can consider that the problem instances of our workbench have been partially stochastically generated.

With this generation procedure, we have defined three different scenarios depending on the constraints used:

- Normal: MaxDT=22 time units (tu), MJ=30 tu, and MTB=180 tu (a time unit = $\frac{1}{2}$ hour).
- Relaxed: only considering overlapping constraints
- Harder: MaxDT=18 tu, MJ=25 tu, and MTB=180 tu

By default, the methods has been tested using the normal scenario, and then, the remainder scenarios have been also used to analyze other possible method behaviors.

## 3 Constraint-based approaches

Constraint-based approaches model the problem as a Constraint Satisfaction Problem (CSP) by means of variables, their domains, and constraints that express dependencies among variable assignments. A solution in this approach is an assignment of a single value from its domain to each variable so that no constraint is violated (Dechter, 2003; Apt, 2003). A problem with a solution is termed *satisfiable* or *consistent*. A *SAT Problem* consists of a CSP with Boolean variables, that is each variable maintains two possible values (Rossi et al., 2006).

All optimization problems are constraint satisfaction problems in the general sense (Rossi et al., 2006). Thus a Constraint Optimization Problem (COP) is defined as a CSP together with an optimization function which maps every solution to a numerical value. The goal is thus to find the solution with the best (maximum or minimum) value.

Constraints methods studied in our analysis are organized in three groups (see Figure 1): systematic, constraint propagation, distributed methods and mathematical optimization. Systematic methods comprises chronological backtracking, branch and bound, and constraint logic programming, in which the flexibility and expressiveness of constraints is enhanced (Barták, 1999; Garcia de la Banda et al., 1996). Constraint propagation techniques are combined with systematic search methods in various forms to reduce the search space (Dechter, 2003; Apt, 2003). Forward checking is the easiest example of this kind of hybrid method. The third group includes the new trends in distributed backtracking algorithms (Yokoo and Hirayama, 2000) and, finally, the fourth category corresponds to the classical mathematical optimization methods.

### *3.1 Chronological backtracking*

This is the simplest search algorithm. This algorithm explores the search tree for all possible assignments alternatives according to a depth-first strategy. At each step, a node is expanded at the lowest level in the tree, meaning that a value is assigned to a variable in which constraints are satisfied (partial solution). This process is repeated until either a complete solution is found or a failure arises, that is, no assignment is possible. Then, the algorithm returns to a higher level at which one resumes the node expansion (Apt, 2003; Dechter, 2003) (see Figure 2). Some heuristics can be applied in order to sort the variables and values to be assigned first, as well as the constraint to be checked first. When a solution is found, the cost of the solution is computed and the process continues. So the complete search space is explored in order to look for other alternative solutions with a lower cost.

### *3.1.1 Problem modeling*

We have formulated the service approach of our problem in terms of variables, domains and constraints as follows: services are our variables, $x_i$; the domain of each variable is the set of drivers $D$. So when a variable has a value assigned $x_i = d_j$, it means that service $s_i$ has the $d_j$ driver assigned to it. Regarding heuristics, we have used the following:

- sorting variables: services have been ordered according to their initial start time (same as in chronological backtracking)
- sorting values: drivers to be assigned to each variable have been ordered according to their cost; so drivers with lower cost (basic, per hour, and per kilometer) are tried first.
- sorting constraints: overlapping constraint, driving time, journey length and cumulated driving time.

Constraints are modeled as follows:

- Overlapping $\forall i, j,\ x_i = x_j$, then $ft_i < st_j + duration(s_{i-j})$ or $ft_j < st_i + duration(s_{j-i})$ where $duration(s_k) = ft_k - st_k$.
- Maximum driving time: $MaxDT \geq TSP_i + \sum_{j \in duties(d_i)} duration(s_j) + TFP_i$ where $TSP_i$ is the time required for driving from the starting driver position to the starting location of the first service; $duties(d_i)$ are the already assigned services to the driver in a partial or candidate solution, and $TFP_i$ is the time required for driving from the final location of the last service to the final position of the driver.
- Maximum journey length: $MJ \geq TJ_i$ where $TJ_i$ is the journey duration of driver $d_i$. It is computed as $TJ_i = (ft_{last(duties(d_i))} + TFP_i) - (st_{first(duties(d_i))} - TSP_i)$ where the first term computes the initial time in which the driver starts his/her duties and the second term, the final time.
- Maximum driving time per two weeks: $MTB \geq hw_i + TJ_i$.

### *3.1.2 Results*

We implemented the algorithm in Delphi and ran the different problems of the workbench. The results obtained with simple backtracking are shown in Figure 3 (see CB line). The x axis shows the complexity of the problem (that is, when solving the 1 service, 1 driver case up to solving the 12 service, 12 driver case of the workbench), while the y axis provides the time in milliseconds. As it is possible to see in the graphics, time increases exponentially and no test have been performed for cases beyond 12. We have obtained similar results in each

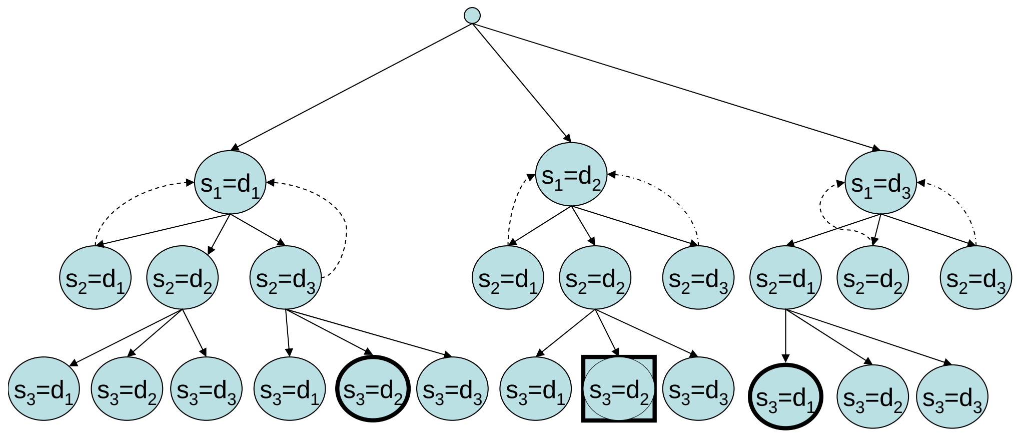

Fig. 2. Search space of the chronological backtracking approach

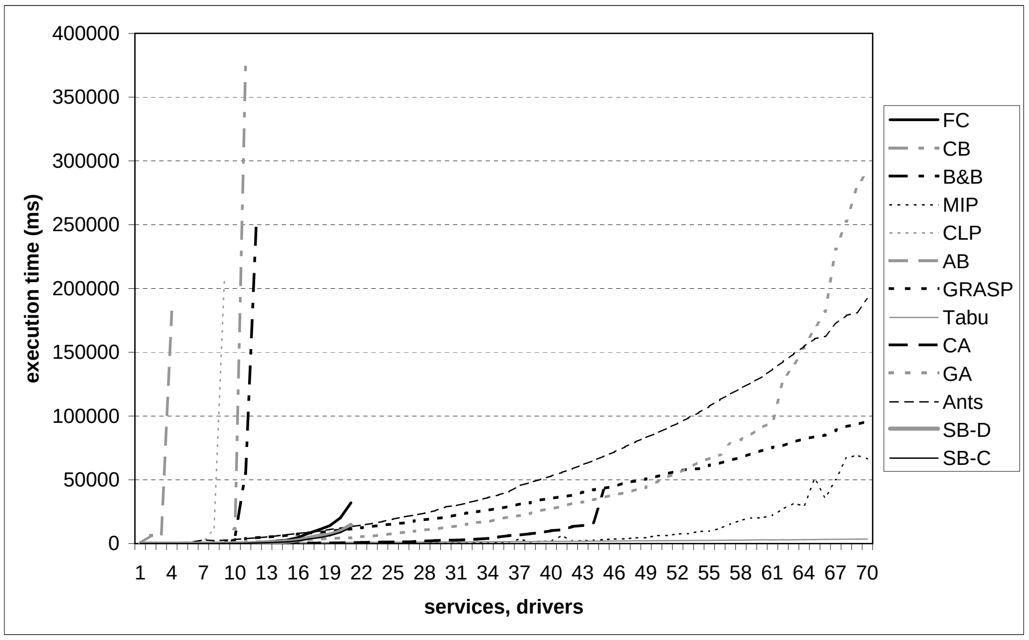

Fig. 3. Times (msec) required by the different techniques analyzed.

scenario of the workbench.

Regarding expressiveness, in chronological backtracking constraints are coded so any constraint of a problem can be programmed. Input data to the program are the services, drivers, locations, and constants used in the constraints (that is, MaxDT, MJ, and MTB). The inputs and the results are easy to provide and interpret by a programmer.

## *3.2 Branch and bound.*

Whenever a partial solution is found, instead of traversing all the search space, the branch and bound method computes its cost and compares it with the best solution found so far (upper bound). If the cost of the partial solution is higher, the algorithm backtracks, pruning the subtree below it (Dechter, 2003). In addition, if the cost of the partial solution is not higher, but it is possible to estimate its final cost, and the estimation goes over the upper bound, it is also pruned. So the key issue in this kind of approach is to define the appropriate estimator.

### *3.2.1 Problem modeling*

The modeling of the problem regarding variables and constraints is the same as in the chronological backtracking method (service approach). The main difficulty of using the branch and bound method was to define the appropriate function to estimate the cost of a partial solution in order to prune the search and provide an answer in a reasonable time (thus improve the results obtained from the chronological backtracking method). This estimation function should take into account the remaining assignments to be performed that depend on both the requested services and the intervening services. This function can underestimate the real cost, but never overestimate it, in order to assure that we are not pruning optimal solutions.

The estimated cost function $F^e$ has been defined as the sum of the individual estimation cost $f^e$ of the remaining services, $R$; that is,

$$F^e(R) = \sum_{s_i \in R} f^e(s_i) \tag{3}$$

The individual estimation of a remaining service $f^e(s_i)$ is based on the minimum driver cost to cover the service. According to individual driver cost of equation 1 two different situations can conflict on the minimum: the driver with the minimum cost $d_c$, or a driver with the minimum distance to the start location of the service $d_d$. In order to solve this conflict, the distance to be covered by both drivers is analyzed, and the minimum one is selected as the value of $f^e(s_i)$.

### *3.2.2 Results*

The results of the branch and bound method can be seen in Figure 3 (see B&B line). We also tested the algorithm when constraints are relaxed or, conversely, are harder, where the algorithm shows a similar behavior. The results are slightly better than in chronological backtracking, but the design (modeling) and the implementation effort were higher. That is to say, the estimator and the search strategy definition took longer to achieve than the naive approach of chronological backtracking.

The branch and bound method implemented here is partially anytime. In other words, if the algorithm is stopped at any moment, it provides the best solution found so far. But it cannot resume its execution.

## *3.3 Constraint propagation*

Constraint propagation algorithms deal with the search space reduction through an inference process which reduces variable values (Apt, 2003). There are two basic schemas: look-back and look-ahead (Barták, 1999). The former checks among already instantiated variables and solves the inconsistency when it occurs. The latter schema is proposed to prevent future conflicts.

Forward checking (FC) is the easiest schema of a look-ahead strategy. When a value is assigned to the current variable, any value in the domain of a future variable which conflicts with this assignment is temporarily removed from the domain (Barták, 1999).

### *3.3.1 Problem modeling*

In this model, we have considered the SAT formulation of the service approach of the problem, that is, each variable maintains two possible values $\{0, 1\}$. Particularly, our variables are the product of drivers and services, as follows:

$$\forall\ d_i\ \epsilon\ D,\ \forall\ s_j\ \epsilon\ S,\ \ X_{ij}$$

where:

- $X_{ij} = 0$ : service $s_j$ doesn't allocate to driver $d_i$.
- $X_{ij} = 1$ : service $s_j$ allocates to driver $d_i$.

According to the SAT formulation, constraints are modeled as follows:

- Each service is allocated to only one driver:

$$\forall\ s_j \,\epsilon\, S,\ \sum_{d_i\ \epsilon\ D} X_{ij} = 1$$

- Overlapping. If $s_j$ and $s_k$ are services with overlapping times: $\forall\, d_i \,\epsilon\, D$, $X_{ij} + X_{ik} \leqslant 1$

- Maximum driving time: $\forall \ d_i \ \epsilon \ D, \ MaxDT \geqslant TSP_i + TD_i + TT_i + TFP_i$ where
  - · $TSP_i$: driving time from the start point of the $d_i$ driver to the start location of his first service.

$$TSP_i = \sum_{s_j \ \epsilon \ S} (Tsp_i sl_j * \max((X_{ij} - \max(\min(\alpha, 1), 0)), 0))$$

where parameter $Tsp_i sl_j$ is the driving time from the start point of the $d_i$ driver to the start location of the $s_j$ service, and $\alpha$ is the sum of all services whose start time < start time of $s_j$.

  - · $TD_i$: driving time dedicated to the services allocated to the $d_i$ driver.

$$TD_i = \sum_{s_j \ \epsilon \ S} (X_{ij} * TS_j)$$

where parameter $TS_j$ is the driving time dedicated to the $s_j$ service.

  - · $TT_i$: driving time required for the intervening services of the $d_i$ driver.

$$TT_i = \sum_{s_j, \, s_k} Tfl_j sl_k * \max(\min(X_{ij}, C_{ik}) - \max(\min(\beta, 1), 0), 0)$$

where parameter $Tfl_j sl_k$ is the driving time from the final location of the $s_j$ service to the start location of the $s_k$ service, and $\beta$ is the sum of all services ranged between final time of $s_j$ and start time of $s_k$. **Falta** $C_{ik}$

  - · $TFP_i$: driving time from the final location of the last service of the $d_i$ driver to his final point.

$$TFP_i = \sum_{s_j \ \epsilon \ S} (Tfl_j fp_i * \max((X_{ij} - \max(\min(\delta, 1), 0)), 0))$$

where parameter $Tfl_j fp_i$ is the driving time from the final location of the $s_j$ service to the final point of the $d_i$ driver, and $\delta$ is the sum of all services whose start time > start time of $s_j$.

- Journey length:

$$\forall \ d_i \ \epsilon \ D, \ MJ \geqslant TJ_i$$

where $TJ_i$ is the journey time of the $d_i$ driver,

$$TJ_i = \max_{\forall \ j}((ft_j + Tfl_j fp_i) * X_{ij}) -$$

$$\min_{\forall \ j}((10001 - (X_{ij} * 10000)) * (1 - C_{ij} + st_j - Tsp_i sl_j))$$

being $ft_j$ the final time of the $s_j$ service, and $st_j$ the start time of $s_j$ (see Definition 1).

- Maximum driving time per two-weeks.

$$\forall \ d_i \ \epsilon \ D, \ MTB \geqslant hw_i + TJ_i$$

where $hw_i$ is the number of hours that the $d_i$ driver has driven during the last two weeks (see Definition 1).

### *3.3.2 Results*

We have used a well-known CSP solver to evaluate our model by means of Forward Checking (FC)[2]. Figure 3 shows the computational time required to solve the problem (see FC line). As it is possible to compare with the previous branch and bound approach, the inclusion of constraint propagation techniques improves the results. Now it is possible to solve up to the 20th case in a reasonable computational time. However, modeling the problem as SAT is not so easy, and require some additional modeling skills.

## *3.4 Distributed approaches. Synchronous backtracking*

In a distributed constraint-based approach, variables and constraints are distributed among automated agents (Yokoo et al., 1998; Yokoo and Hirayama, 2000). In (Yokoo and Hirayama, 2000), Yokoo and Hirayana present a formalization and algorithms for solving distributed CSPs, classify them as either synchronous backtracking or asynchronous backtracking and differentiates them from previous centralized methods (Yokoo and Hirayama, 2000). In a centralized approach a single agent keeps all the information about variables, their domains and constraints, and solves the problem using classical constraints algorithms (such as the chronological and branch and bound methods). In a distributed approach a set of agents are committed to a set of subproblems in order to solve the global problem. These agents can work in a synchronously or asynchronously way. In a synchronous approach, agents agree on an instantiation order for their variables. Each agent, receiving a partial solution from the previous agent, instantiates its variables based on the constraints that it knows about. If it finds such a value, it adds this to the partial solution and passes it to the next agent. Conversely, it sends a backtrack message to the previous agent (Yokoo et al., 1998; Hirayama and Yokoo, 1997). In an asynchronous approach, each agent runs concurrently and asynchronously (see next section).

In this section we deal with the synchronous approach following the distributed framework of (Salido and Barber, 2006) in which the problem is partitioned in $k$ subproblems which are as independent as possible, the subproblem are classified in the appropriate order and they are solved concurrently.

[2] Forward Checking was obtained from CON'FLEX. It can be found in: http://www.inra.fr/bia/T/rellier/Logiciels/conflex/.

### *3.4.1 Problem modeling*

The distribution of the road passenger transportation problem can be carried out by means of problem topological properties (number of variables) or by means of problem size (number of constraints). In this work, we tested both kinds of distributions:

- *DCSP by Constraints.* Each agent concentrates the constraints of the same type. Thus, the first agent is committed to solving the CSP restricted to allocating and overlapping constraints; the second agent works on the constraints related to maximum driving time; the third agent works on the journey length constraints; and the fourth agent works with the constraints related to the maximum driving time per two weeks.
- *DCSP by Drivers.* Each agent is committed to assigning values to variables related to a driver. Thus, many related variables are grouped in the same subproblem.

Regarding the method used in each isolated agent, we followed the FC method of the previous section.

### *3.4.2 Results*

Figure 3 shows the computational time required to solve the problem using the synchronous backtracking approach both in the *DCSP by Constraints* and *DCSP by Drivers*) distributions (see the SB-C and SB-D lines correspondingly). As it is possible to observe in the plot, the run-time in all instances was better in the distributed models than in the corresponding centralized FC model (see FC line). It must be taken into account that the number of variables and constraints in the proposed model is large and the complexity grows exponentially. However, the distributed models behaved more consistently than the centralized model in all instances.

Figure 3 also shows that the synchronous backtracking by constraints also exhibited better behavior than the synchronous backtracking by drivers. This is due to the fact that in the constraint distribution there are fewer subproblems (four) than in the driver distribution (as many as drivers). In this way, if there are many drivers, the communication between agents becomes hard, and the computational cost increases. Obviously, the required computational memory in the distributed model is lower that in the previous FC case. Although each agent performs its own FC process, the problem size is distributed between all agents.

Asynchronous backtracking was one of the first algorithms to cope with distributed CSP (Yokoo and Hirayama, 2000; Yokoo et al., 1998). In this method, agents act asynchronously without any global control. Each agent instantiates a single variable and communicates its value to the agents with connecting links. Two agents are neighbors if the variables they control have some constraint/s. Thus each agent is only aware of the constraints associated to the variables it controls. One of the requirements of the system, then, is to have a problem in which locality holds; that is, the set of variables can be partitioned in such a way that constraints can be managed locally.

Asynchronous backtracking has been generalized to solve DCOP which includes an objective function so that agents coordinate in order to optimize it (Modi et al., 2005). One of the algorithms proposed in the literature is ADOPT, in which the strategy used to find the solution is called opportunistic best-first search (Modi et al., 2005). In this strategy, agents are organized in a tree structure that establishes a priority among them (parents to children); that is, constraints are only allowed between an agent in its ancestors or descendants. The priority is used to guide the backtracking process from the lower levels to the upper ones. All agents begin to set their variables with an initial value and send this assignment to the lower levels. When some agent cannot perform an assignment, it asynchronously sends a *nogood* message to its ancestors.

### *3.5.1 Problem modeling*

In this model we followed the service approach, so each variable represents a service. Then, each variable is assigned to an agent. The cost function is distributed among variables. So, given a pair of variables $s_i, s_j$, the cost is represented as a "soft" constraint of compatible (good) values and its cost. For example, constraint $c_{i,j} = (k, l, m)$ means that when $s_i = k$ then variable $s_j$ can be set to $l$; and the cost of this assignment is $m$. Thus, the domain of the variables cannot be directly the drivers as expected, since the cost associated to a "soft" constraint does not depend only on the two services (variables) but on all the services assigned to the same driver in a journey. Therefore the domain of the variables is actually in $[1, ...nc]$, where $nc$ is the number of combinations of journeys assigned to drivers ($nc < journeys * drivers$). If variable $s_i$ is set to $j$, it means that service $i$ appears in the solution in the $j$ assignment (journey and driver).

Regarding "hard" constraints (the primal constraints in our problem definition), they are represented by an $\infty$ cost. The total number of constraints required is $n^2 * nc$, where $n$ is the number of variables (services) and $nc$ the number of journey combinations as above.

### *3.5.2 Results*

Thanks to the fact that ADOPT can be used under the GNU licence, we have had the opportunity to test it in our problem. We cannot test the system from more than four variables, as shown in Figure 3 (see AB line). The number of constraints requiring specification is huge, and so the algorithm gets rapidly collapsed. These results are not surprising since ADOPT was originally designed to deal with CSP instead of DCOP problems. Recent modifications of the algorithm, such as (Ali et al., 2005), could improve the results.

## *3.6 Mathematical optimization. Mixed integer programming*

Mixed integer programming (MIP) is probably the most important technique in the field of Operational Research (Maroto et al., 2003; Hoffman and Padberg, 1996). In this technique, problems are represented by mathematical models in which the objective function is linear and the constraints are given by linear equations and inequalities, and so, variables are necessarily numerical. If the domain of the variables are integer, we are dealing with integer programming. When dealing with integer and real variables, mixed integer programming methods are required. Furthermore, if the relationship among variables cannot be expressed by a linear function, then non-linear programming methods are necessary (Cooper and K. Farhangian, 1985). Other approaches, such as stochastic programming, can also be found in the literature. See (Orden, 1993) for a review.

### *3.6.1 Problem modeling*

In order to model our problem in the MIP paradigm, all information about the problem should be known. In this sense, intervening services should be known in advance, conversely to the service approach of the branch and bound or chronological backtracking method, which can be generated while solving the problem. Since the journey approach contemplates a complete simple formulation, we have adopted it for modeling the problem in linear programming.

Then, the variables required to model our problem are the following:

- Data: drivers, journeys and services

- Parameters: cost of the journeys per driver (co), and services included in journeys (mat).
- Decision variables: journeys included in the solution (sol), and drivers assigned to those journeys (solution). The linear programming method should provide the values for these variables, so they are the solution to the problem. These decision variables are binary (for example if journey $j$ is included in the solution the value of corresponding decision variable is 1, otherwise 0). Consequently decision variables are integer values $\in \{0, 1\}$.
- Objective function to be minimized:

$$\sum_{d \in D, j \in J} (co[j,d] * solution[j,d]);$$

- Constraints

(1) Each journey should appear *at most* once in the solution

$$\forall j \in J \sum_{d \in D} solution[j,d] \leq 1$$

(2) Each driver should be assigned *at most* once to a journey

$$\forall d \in D \sum_{j \in J} solution[j,d] \leq 1$$

(3) Each service should appear *at most* once in the solution (so journeys including the same services are incompatible)

$$\forall s \in S \sum_{j \in J} mat[j,s] * sol[j] = 1$$

(4) Each journey included in the solution should have a driver assigned

$$\forall j \in J \sum_{d \in D} solution[j,d] = sol[j]$$

It is interesting to note, then, that MIP allows the definition of the constraints outside the code of the algorithm, conversely to the branch and bound methods, in which constraints are coded. However, it requires a lot of data (journeys, matrix of journeys and costs, matrix of journeys and services). So for a large problem, a preprocessing step is required in order to generate all these data.

### *3.6.2 Results*

According to their popularity and the benefits that MIP provides to industries and companies, several tools have been developed. In particular, CPLEX [3] is

[3] ILOG CPLEX: http://www.ilog.com/products/cplex/.

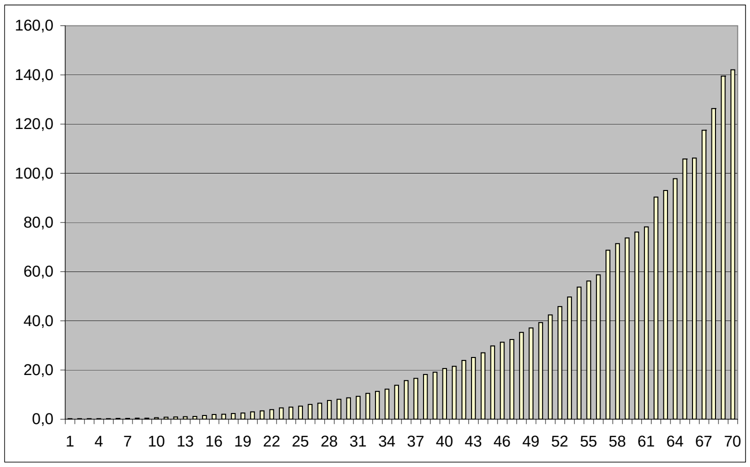

Fig. 4. Memory in MB required to store the input files for GLPK.

one of the best tools found on the shell. Recently, GLPK [4] has been developed under the GNU licence with an efficiency close to CPLEX. We have selected GLPK to cope with our experiments.

Figure 3 shows the computational time required to find the optimal solution using GLPK (see MIP line). MIP is able to solve all the problems in our workbench, up to the last one in 32 seconds. In addition to this time, a preprocessing time is required to generate the journeys and other GLPK inputs (data, parameter, variables), summing up a total amount of 66 seconds for the 70th case. We have carefully analyzed this results to understand such a good behavior. In order to solve the integer programming problem, GLPK first finds a relaxed solution to the problem with continuous variables, and then it finds the closest integer solution to the original problem. In our case, we have realized that the continuous solution coincide almost all the times with the integer one. And this is the cause of obtaining so efficient results.

Nevertheless, we should also analyze other features of MIP. For example, the amount of memory required to solve the problem (space complexity). Figure 4 shows the memory required in each experiment. If we need to extend the problem, we need to review our model, defining new parameters and data that could increase the space complexity and the preprocessing time. In addition, the interpretation of both the model and the solution provided by this technique is not always straightforward when the problem becomes more complex.

[4] GLPK, GNU Linear Programming Kit, Free Software Foundation, http://www.gnu.org/software/glpk/.

Constraint logic programming (CLP) is a general purpose paradigm that deals with complex problems by means of the power of constraints solvers and the versatility of the declarativity of logic programming (Jaffar and Maher, 1994; Hentenryck, 1989). Unfortunately, as we will see, this "general purpose" nature becomes a weakness for our problem, in comparison to specialized solvers like MIP techniques. The most popular language that implements the CLP paradigm is Prolog, although there are several other good approaches (see Ferández and Hill (2000) for a comparative survey). Among them, we have selected `GProlog`[5], which includes a finite domain constraint solver whose performance is comparable to other commercial systems[6].

The classical scheme of a CLP program consists of first creating the variables of our model and assign them a domain, then constraining the variables depending on the requirements of our problem and finally, asking for an assignment to the variables, in accordance with their domains, that satisfies all their constraints. The intrinsic Prolog backtracking allows us to enumerate all solutions. In our case the *variables assignment* part must deal not only with the constraints but also with an optimization requirement.

### *3.7.1 Problem modeling*

We decided to use the journey approach because the translation from the MIP implementation is quite immediate. In this approach the possible journeys and cost are already precalculated. We could have used the service approach if our problem is subject to further new constraints introductions or modifications, but for the purpose of our exploratory work we believe that the journey approach is convenient. As we will see, the Prolog code that we propose is very concise and comprehensible.

First, we define a list $L = X_1, \ldots, X_{nc}$ of variables with domain $\{0, 1\}$ that correspond to all the combinations of journeys assigned to drivers. $X_i = 1$ means that the combination is selected, while $X_i = 0$ means that it does not. Then, the variable *Cost* is the sum of $X_i * c_i$'s, where $c_i$ is the precalculated cost associated to the journey and the driver of the $i$ combination. So the problem code starts as follows:

[5] GNU Prolog, Free Software Foundation, http://www.gnu.org/software/gprolog/.
[6] We also tried SICStus Prolog, but `GProlog` performance was slightly better.

```
transports([ $Cost | L$ ]) :-
    $L$ = [ $X_1, \ldots, X_{nc}$ ],
    fd_domain( $L$, 0, 1 ),
    $Cost$ #= $c_1 * X_1 + \ldots + c_{nc} * X_{nc}$,
```

We have to impose the constraints of the journey approach model to the variables. First of all, we require that *each driver can do at most one journey*: for each driver $d$ we take the variables $X_d^1, \ldots, X_d^{d_j}$, corresponding to all their possible $d_j$ journeys, and force them to be all equal to 0 but one, which can be equal to 1. To do so we use the finite domain predicate `fd_atmost(N,List,V)` which posts the constraint that at most `N` values of `List` are equal to `V`: `fd_atmost(` 1, $[X_d^1, \ldots, X_d^{d_j}]$, 1 `)`.

Now, we require that *each service must be done by exactly one of the journeys proposed by the solution*: for each service $s$, we take the variables $X_s^1, \ldots, X_s^{s_t}$, corresponding to all the $s_t$ journeys that include the service, and force them to be all equal to 0 but exactly one. To do so we use the finite domain predicate: `fd_exactly(N,List,V)` which posts the constraint that exactly `N` variables of `List` are equal to the value `V`. That is: `fd_exactly(` 1, $[X_s^1, \ldots, X_s^{s_t}]$, 1 `)`.

Finally, we ask the constraint solver of `GProlog` to get an assignment to the list of variables $L$ so that it minimizes the value of the variable $Cost$, i.e. the cost of the journeys. We do so using the following finite domain predicates:

- `fd_labelling(Vars,Options)`, which assigns a value to each variable of the list `Vars` satisfying all the constraints that the variables may have. The `Options` parameter allows us to control the way in which the assignments are obtained. In our application, the option `value_method(max)`, that forces the solver to enumerate the values from greater to smaller, has been empirically crucial;
- and the optimization predicate `fd_minimise(Goal,X)` that repeatedly calls `Goal` to find a value that minimizes the variable `X`. In fact, this predicate uses a branch and bound algorithm with restart.

The resulting combination of both predicates is the following: `fd_minimize( fd_labeling(`$L$`, [value_method(max)]),` $Cost$ `)`.

### *3.7.2 Results*

Results on Prolog have been uncourageous. The computational time obtained for the tests applied leads us to stop the experimentation in the 10th example (see CLP line in Figure 3). It is one of the worst obtained in our experiment. In addition, Prolog only allows a solution to be obtained when it finishes, that is, it does not exhibit an anytime behavior.

In our opinion, CLP could be a helpful tool for a dynamic problem where constraints evolve and versatility is a crucial point. In a work like this where the constraints and the model are so fixed, specialized tools like the MIP system take advantage of their specialization with a very powerful (linear) constraint solver and beat general purpose paradigms like CLP.

## 4 Metaheuristics

The word metaheuristic was coined by Glover in 1986 (Glover, 1986) and its meaning has been changing since then. In the original definition, metaheuristics are methods that combine local improvement procedures and higher level strategies to create a process capable of escaping from local optima and performing a robust search for a solution space (Glover and Kochenberger, 2003). Nowadays, metaheuristics can be seen as intelligent strategies to design or improve heuristics procedures with a high performance generally combining constructive methods, local search methods, concepts that come from Artificial Intelligence, biological evolution and statistics methods (Melián et al., 2003). In this paper we have analyzed GRASP and Tabu search as representative methods. Genetic algorithms can also be found in this category depending on the information source consulted.

### *4.1 GRASP*

Greedy Randomized Adaptive Search Procedure (GRASP) was developed by Feo and Resende (Feo and Resende, 1995). It is an iterative procedure with two phases: the first constructs an initial solution using a randomized greedy function; and the second improves the quality applying a local search procedure. The best overall solution is kept as the result.

In the first phase, a feasible solution is constructed iteratively. All the elements are ordered in a candidate list with respect to a greedy function, which measures the benefit of selecting each element. The list of best candidates is the *restricted* candidate list (RCL). The factor $\alpha$ determines the quality of the solutions in the RCL; if $\alpha$=0 only the best candidate is in the RCL making the algorithm pure greedy; on the other hand, if $\alpha$=1 all the feasible candidates are in the list. One candidate of the list is chosen randomly. It is said that the heuristic is adaptive because the benefit associated with every element is updated after the selection of the candidate at every iteration to reflect the changes brought about by the selection. Using this technique different solutions are obtained at each GRASP iteration.

The solutions generated by the construction phase do not guarantee they are locally optimal with respect to a simple neighborhood; therefore, it is good to apply a local search to attempt to improve the solution constructed. A local search algorithm works in an iterative way by replacing the current solution by a better one in the neighborhood. It finishes when no better solution is found.

### *4.2 Problem modeling*

We have formulated the problem using the service approach. For the constructive phase all the services are ordered by departure time. A list of all possible drivers that can perform a service is made and the cost of assigning the service to that driver is calculated. The RCL is built accordingly with the randomize factor $\alpha$, that was set to $\alpha = 0.1$ after 20 trials. A driver from the RCL is selected randomly and all the variables are updated. We follow this procedure until all the services have a driver assigned to them.

The local search attempts to reduce the cost by reducing the number of assigned drivers. In order to do that, the algorithm looks for drivers that perform only one service and finds out if another driver could do it.

### *4.3 Results*

GRASP, solve all the problems on the workbench, so the solutions found by this technique satisfy the constraints (see GRASP line in Figure 3). However, the objective function value is not the optimal but close enough. We have measured the percentage of average deviation over the optimal solution in the 200,000 solutions generated: 5.07 % (with $\sigma = 0.015$). Similar results were obtained when relaxing the problem or adding harder constraints. In all cases, the required computational memory is no relevant.

### *4.4 Tabu search*

Tabu Search (TS) was introduced by Fred Glover in (Glover, 1986) and its main characteristic is the use of adaptive memory which allows to explore different regions in the search space (short term memory) and to intensify the search in promising areas (long term memory). Advanced features can be found in (Melián et al., 2003).

When implementing a **basic** TS procedure, the first thing to do is to construct an initial solution ($s$). Then, a neighborhood $\mathbf{N}(s)$ is constructed to find adjacent solutions that can be reached from the actual one. A *move* leads from one solution to the next in the neighborhood. The *tabu structure* records a subset of the possible moves in the neighborhood as forbidden (Tabu) because they were made in the recent past. Therefore, when doing the local search, a move that it is not tabu is found.

The move can improve or unimprove the value of the objective function, and the best overall solution is kept as the result. Figure 5 shows a pseudocode of this procedure. A comprehensive examination of this methodology and advanced features can be found in (Glover and Laguna, 1997).

```
procedure TABU_SEARCH
 choose $x \in \mathbf{X}$;
 $x^* \leftarrow x$;
 do {
   $iter \leftarrow iter$ + 1
   Find $x' \in \mathbf{N}(x) \setminus \mathbf{T}(x)$ such that $f(x')$ is minimized;
   $x \leftarrow x'$;
   Update $\mathbf{T}(x)$;
   if($f(x) < f(x^*)$ {
    $x^* \leftarrow x$;
    iter_best$\leftarrow iter$; }
 } while(stopping criterion not satisfied)
 Return $x^*$;
end TABU_SEARCH
```

Fig. 5. Basic Tabu Search Procedure.

### *4.4.1 Problem modeling*

We have formulated the problem using the service approach. The initial solution is constructed using the same procedure described in Section 4.1 using $\alpha = 0$. One of the key points is the definition of the neighborhood for the problem in consideration. We propose a exchange neighborhood, i.e. remove a driver assigned to one service and add a new one to cover it. The neighborhood considers all solutions that can be obtained from the current solution by the exchange of drivers.

In this implementation, a *move* consists in changing one service from the actual driver to a new driver. The tabu list keeps the record of the services moved. To make a move, drivers are ordered increasingly by the number of services that they have been assigned. The procedure tries to find an active driver that can do the service. A move can be carried out if all the constraints for the new driver are satisfied after the change and the move is not in the tabu list. The value of each solution is the total cost of the drivers.

The size of the tabu list was determined experimentally with 20 instances and it was chosen a size of $n = 90$ % of the number of services in the problem, which means that a move is in the tabu list during $n$ iterations. For each problem one initial solution was built and 200,000 moves were performed. The best overall solution is presented as the solution of the problem.

### *4.4.2 Results*

The solutions found by this implementation of TS are a little more expensive than the ones obtained by the other metaheuristic method (GRASP): 8.66 % (with $\sigma = 0.30$) of average deviation over the optimal cost. This is due to the simple neighborhood and movements design for the search. However, the algorithm has a lower computational cost, as it is shown in Figure 3 (see Tabu line). In addition, when using harder scenario of our problem, it has less impact on the total computational time that GRASP has. Again, the required computational memory is not relevant.

## *4.5 Combinatorial auctions*

Auctions have been studied in Economics as a mechanism for dealing with shared resources. Among the different types of auctions, combinatorial auctions allow bidders to submit bids on bundles or packages of items (Kalagnanam and Parkes, 2004; Cramton et al., 2006). Given a set of items $I = it_1, ..., it_n$, each bid $b_j$ is characterized by the subset of items $g(b_j) \subset I$ that the bidder (agent) requests, and its price, $p(b_j)$. Formally: $b_j =< g(b_j), p(b_j) >$.

The auctioneer is faced with a set of bids (price offers) for various bundles of goods and his aim is to allocate the goods in a way that maximizes his revenue, which has been called the winner determination problem (Leyton-Brown, 2003). The winner determination problem (WDP) is known to be a NP-complete problem and there are many approaches that can be used to solve this problem. On one hand there are specific algorithms that have been created exclusively for this purpose, being CABOB(Sandholm, 2002) the one that has been proved to be one of the bests. On the other hand, the *WDP* can be modeled directly as a MIP and solved using a generic MIP solver. Due to the efficiencies of actual solvers like GLPK (free) or CPLEX (commercial), the research community has nowadays mostly converged towards using MIP solvers as the default approach for solving the *WDP*.

### *4.5.1 Problem modeling*

In accordance with our problem, we define drivers as the bidders who are trying to buy services. This view of the problem makes driver constraints be managed locally by each driver and facilitates further extensions of the problem, as adding drivers' preferences on services.

In addition, there is an auctioneer agent that decides upon the allocation by solving the *WDP*. First of all, the auctioneer opens the action by announcing the services to be deployed. Then, drivers submit their bids according to their constraints. And then, the auctioneer provides the final allocation.

Each driver generates as many bids as possible according to all the possible combinations of services (journeys) he can accomplish. The services of a bid ($g(b_i)$) make up a journey and consistently the price ($p(b_i)$) is related to the cost of the journey.

Regarding prices, since the WDP consists in maximizing the outcome of the auction, we develop a mechanism to provide an inverse-like cost function. If $c(b_i)$ is the cost of the $g(b_i)$ services of a bid (i.e. journey), it is not enough to define $p(b_i) = \frac{1}{c(b_i)}$ due to the price additivity, since $\frac{1}{c(b_i)} + \frac{1}{c(b_j)}$ is not the same as $\frac{1}{c(b_i)+c(b_j)}$. Given such a situation, we have defined the inverse-like function as follows:

$$p(b_i) = length(g(b_i)) * maxCost - c(b_i) \tag{4}$$

where $maxCost$ is a value higher than the cost of any bid.

### *4.5.2 Results*

Figure 3 shows the computational time required to find the optimal solution (see CA line); this time includes bid generation, and the time required to solve the WDP (including the MIP model generation and the GLPK time). We are able to solve up to 45 cases before the problem becomes untractable due to memory constraints. Even that MIP was efficient to solve the original problem formulated as the journey approach (see Section 3.6), we have already detected an exponential increase of the memory required by the method (see Figure 4). In the combinatorial auction approach, the number of journeys is multiplied by the number of drivers, so the MIP model that results from the WDP is $n$ times higher than the original MIP formulation. As a consequence, GLPK memory collapses after the 45 problem.

# 5 Bioinspired approaches

Evolutionary and bioinspired approaches are based on using analogies with natural or social systems to design non-deterministic and heuristic methods for searching, learning, behavior, etc. Evolutionary algorithms make use of computational models of natural processes of evolution of individual populations through selection and reproduction processes. These approaches include genetic algorithms, population-based heuristics, memetic algorithms related to cultural evolution, etc. Recently, another group of proposals inspired by biological models has arisen, such as those based on colonies of ants, societies or clusters (hives), the immune system, self-organization or artificial life, etc.

Evolutionary and bioinspired approaches allow addressing the resolution of a great variety of problems of optimization and searching in complex spaces. Due to the scope of these methods, they are also related to metaheuristic methods, such as Tabu search.

For our exploratory work, we have selected genetic algorithms and ant colonies methods.

## *5.1 Genetic algorithms*

Genetic Algorithms (GA) were first introduced by John Holland in 1975, and are inspired on evolution rules. The main idea is that during evolution the best fitted individuals have greater probabilities to survive and reproduce, while the least fitted will be eliminated.

Main features in GA are the codification for the $n$ individuals (population), and the fitness, selection, crossover and mutation functions. Initial population (p) is usually built of feasible solutions. Each individual of the population can be seen as a solution, and the genetic information can be expressed as a binary vector where the solution is encoded. The evaluation function assigns a value (*fitness*) to each individual, usually as a measure of its quality. The *Selection Function* determines which individuals will generate the new ones. Different types of selection directly or indirectly use the fitness value to guide the procedure to find better solutions (Alba et al., 2003). The *Crossover Operator* interchange the information between parents and the *Mutation Operators* modify the information of the individual in order to introduce diversity into the population.

*Batch update* replaces the initial population completely with the new population, and is only performed when all the individuals have been generated.

To avoid the loss of the best individual, batch update usually transfers the best individual in the initial population into the new generation (*elitism*). The process of generating new population is repeated either for a finite number of iterations or until some given condition holds.

### *5.1.1 Problem modeling*

We model the problem using the journey approach. This allowed us to divide the main problem in two subproblems: first the assignment of the journeys to the solution and, second, the allocation of drivers to journeys. For the first subproblem we used a GA and for the second one a greedy function.

To solve the first subproblem, each individual of the population codifies the set of journeys, having as many bits as journeys in the problem. If the bit value is 1, the journey is in the solution, and 0 otherwise. To build the initial population, journeys are selected randomly until all the services are covered.

The evaluation function counts the number of journeys that each individual has, assigning that value as the individual fitness. The selection function is an inverted roulette where individuals with fewer journeys are more likely to be selected for crossover.

As the crossover operator we used the *fusion* operator proposed in (Beasley and Chu, 1996), which produces only one offspring and selects the offspring bit values based on the fitness of the parents. Let $f_{P_1}$ and $f_{P_2}$ be the fitness of the parents $P_1$ and $P_2$ correspondingly, and let $C$ be the offspring. Then, $C$ is generated as follows: for all $i = 1, \ldots, n$

(1) if $P_1[i] = P_2[i]$, then $C[i] = P_1[i] = P_2[i]$
(2) if $P_1[i] \neq P_2[i]$, then
   (a) $C[i] = P_1[i]$ with probability $p = \frac{f_{P_1}}{P_1 * P_2}$
   (b) $C[i] = P_2[i]$ with probability $1 - p$

Mutation function is the standard bit-flip. After crossover and mutation, individuals may violate the problem constraints (i.e. some services are not covered). A repair operator was designed to make all solutions feasible. Finally, we chose a batch population update using the elitism operator.

The individual returned by the GA only has the journeys in the solution. Thus, we eliminate the services that are covered by more than one journey, by randomly selecting a journey in the solution that covers each one of them. Finally, to assign the driver, a greedy function finds the lowest cost driver for the first journey, the second lowest for the second journey and so on. This is possible thanks to the journeys structure that follows a decreasing pattern: the first journeys have more services than the last ones.

### *5.1.2 Results*

First of all, we ran twenty experiments to determine the crossover probability to 90% and mutation probability to 1%. We set the number of individuals and the number of generations to 100.

Regarding the workbench, and in order to take advantage of the random feature of the algorithm, each problem was run 20 times, meaning that for each problem we generated 200,000 individuals. The second part of the solution is applied to the individuals in the last generations that has the best fitness, and the best overall solution is presented as the solution of the problem.

The computing time shown in Figure 3 by this technique (see GA line) improves the results of previous approaches except MIP and GRASP. Regarding the solution costs, however, the GA systematically finds better solutions than Tabu search or GRASP: the average deviation over the optimal solution is 4.09 % ($\sigma = 0.023$).

## *5.2 Ant colony optimization*

Another bioinspired technique for solving combinatorial optimization problems is Ant Colony Optimization(ACO). This technique was proposed by Dorigo and others (Dorigo et al., 1999; Dorigo and Di Caro, 1999) and is inspired by the behavior of ants in order to find food. The ACO algorithm consists of a colony of ants that looks for solutions performing randomized walks on a completely connected graph $G_C = (C, L)$. The nodes belong to a finite set of components $C = \{c_1, c_2, ...., c_n\}$ and every candidate solution $x$ is equal to a sequence of these components $x = \{c_i, c_j, ..., c_h, ...\}$. $G_C$ is called the construction graph and elements of $L$ are called connections. Each connection has a value $\tau_{(i,j)}$ that represents the goodness of using that connection.

We can see the pseudocode of the ACO algorithm in Figure 6. Three main steps are considered. First, the "ConstructAntsSolutions" consists of creating the ant colony, and sending all its individuals to find solutions in the graph (problem). Ants walk the graph. An ant at node $i$ that has walked a partial solution $x_h$ decides the next node to visit $c_{h+1} = j$ based on the probability $P_r(c_{h+1} = j|x_h)$. This probability is computed as follows:

$$P_\tau(c_{h+1} = j|x_h) = \begin{cases} \frac{{\tau_{ij}}^{\alpha}}{\sum_{(i,l)\epsilon N_i^k} {\tau_{il}}^{\alpha}} & \text{if } (i,j)\epsilon N_i^k; \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

where

- $N_i^k$ is the neighborhood of ant $k$ at node $i$; the set consists of all the nodes that the ant $k$ can visit after node $i$;
- $\tau_{(i,j)}$ is the connection strength of the node $i$ with its neighbors (pheromone level); it is initialized randomly and updated throughout the process;
- $\alpha$ is an algorithm parameter.

The ants stop walking the graph when a feasible solution has been constructed or when the neighborhood is empty. In the second case, the ant is useless and is not taken into account for the following steps.

```
procedure ACO()
  while(termination condition not met)do
      ConstructAntSolutions
      UpdatePheromones
      DaemonActions        % optional
  end-while
end Procedure
```

Fig. 6. Basic ACO metaheuristic Algorithm pseudocode.

The second step of the algorithm of Figure 6 consists of updating $\tau_{(i,j)}$. On the one hand, values of the connections that are part of one solution are increased. On the other hand, all connections that do not participate in a solution are decreased (pheromone evaporation). The objective of pheromone evaporation is to avoid a convergence of the algorithm for a suboptimal solution.

Finally, several centralized heuristics can be applied, when required, in the last step of the algorithm, DaemonActions.

These three steps are repeated in the loop until a termination condition is met. This condition can be either a given number of iterations or a convergence criteria, among others. Then, the walk corresponding to the highest $\tau_{ij}$ values constitutes the best solution.

### *5.2.1 Problem modeling*

To solve the problem studied in this article using ACO, we consider the construction graph $G_C$ where $C = S \cup D \cup dummyNode$,which means that the nodes are drivers and services, and there is an additional initial node that has neither a service nor driver. The graph is fully connected, so there is a label $l_{i,j}$ for every pair of nodes $c_i, c_j$. An ant that walks over the link $l_{(i,j)}$ means that the service $i$ is assigned to the driver $j$.

Regarding the first step of the algorithm, the *ConstructAntSolution*, all ants are created at the dummy initial node. Ants decide the next node to visit according to equation 5 with the following modification. The probability becomes 0 when visiting a neighbor node that breaks some problem constraints. Thus, if $i$ is the initial node, the neighborhood are all the services; if $i$ is a service node, the neighborhood are all the drivers that can attend to that service without breaking the problem constraints; finally, if $i$ is a driver node, the neighborhood are the services nodes that haven't been visited by the ant.

For the second step of the algorithm, the *UpdatePheromones*, two updating functions were defined. First, when each of the ants finds a feasible solution in the previous step of the algorithm, $\tau_{ij}$ is updated according to the cost. Regarding the pheromone evaporation, the following function was used:

$$\tau_{ij} \longrightarrow \tau_{ij} * (1 - \rho) \tag{6}$$

where $\rho$ is the evaporation factor and is another parameter of the algorithm. Finally, no special methods have been implemented for the third step of the algorithm (*DaemonActions*).

#### *5.2.2 Results*

Based on a series of experiments, the parameters used were $\alpha = 2$,$\rho = 0.1$ and a colony size equal to 100. Due to the fact that the algorithm is probabilistic, each problem was run 20 times and the best overall solution is presented as the solution of the problem. The time required for the execution of the algorithm is shown in Figure 3 (see ANT line). We have not found the optimal solution in any case; and the solutions found are much worse than in the metaheuristics and genetic algorithms approaches. This could be due to stagnation: the undesirable situation in which all ants repeatedly construct the same solutions making any further exploration in the search process impossible. Recent versions of the algorithm propose several alternatives to avoid it, by combining exploitation and exploration (Maniezzo et al., 2004). Regarding the former, the ants use information of the past found effective solutions in order to choose the node to visit. On the other hand, exploration favors the discovering of new paths, trying to avoid stagnation.

## 6 Discussion

From the experience obtained in our experimental tests of the different methods, several factors have been measured as relevant ones when selecting a technique. In particular, we distinguish the following:

- modeling expressiveness: whether the methods allow the specification of constraints as part of the program (constraints coded), or whether constraints should be provided explicitly (the declarative way) by means of either compatible variable-values pairs or with the use of functions. When constraints should be provided explicitly and the problem is large, some kind of preprocessing step is required to obtain the constraint set even though sometimes the specification language provides some way to specify them by means of complex expressions.

- anytime: whether the algorithm can be stopped and its execution resumed, giving the best solution found so far or not, or only partially (can stop but not resume).
- time complexity: whether the method as tested is able to solve up to the 70th test case (low), up to the 20th case (medium) or few cases (high).
- memory complexity: the amount of memory required by the method, to store either constraints or internal data. High means that the method requires a lot of memory, and dynamic memory or other kind of programming tricks should be used to keep handling memory in an efficient way.
- parameter tuning: whether the method requires several runs in order to tune the parameters required. In this sense, the label ”Yes” indicates that with the current parameter estimations the algorithm has not found the best solution.
- tool: whether there is a free licence tool on the shell to test the problem or not. Note that tool availability could force the problem to be modeled according to the tool requirements. In addition, the available tools are not always the most efficient ones.

In Table 1 there is a summary of the methods analyzed together with a checklist of the properties that they exhibit. Regarding computational time, MIP is the one exhibiting the better behavior. Other recent paradigms like Tabu search, GRASP, genetic algorithms and ant colony optimization are also able to deal with the whole workbench at a reasonable computational effort. In Figure 7 the methods are organized in the three time complexity categories according to the results obtained in our experiments. This does not mean that the optimal solution could be found in either Tabu, GRASP or GA, but a quasi-optimal one (see Figure 8 for a comparison on the cost of the solutions found). Much more effort should be made to find the different parameters that tune the algorithms. In this sense, there is much more uncertainty in the development of the algorithm from an engineering point of view.

Regarding modeling expressiveness, in general, declarative methods are the easiest way to make initial approaches to simple problems. However, with complex problems, with many constraints, search methods (chronological backtracking and branch and bound) and constraint propagation methods have proved to be the easiest way to approach the problem the first time. This has been our case. So, even though better computational times are obtained with mixed integer programming, the mixed integer programming model was

| Method | Modeling | Anytime | Time | Memory | Tuning | Tool |
|---|---|---|---|---|---|---|
| Chronological backtracking | Coded | Partial | High | Low | No | No |
| Branch&Bound | Coded | Partial | High | Low | No | No |
| Mixed integer programming | Declarative | No/Yes | Low | High | No | GLPK |
| Constraint logic programming | Declarative | No | High | Low | No | GProlog |
| Forward checking | Declarative | No | Medium | Medium | No | ConFlex |
| Synchronous backtracking | Declarative | No/Yes | Medium | Medium | No | No |
| Asynchronous backtracking | Declarative | No | High | High | No | ADOPT |
| Tabu | Coded | Yes | Low | Medium | Yes | No |
| GRASP | Coded | Yes | Low | Medium | Yes | No |
| Combinatorial auctions | Coded | Partial | Medium | High | No | GLPK |
| Genetic algorithms | Coded | Yes | Low | Low | Yes | No |
| Ant colony optimization | Coded | No | Low | Medium | Yes | No |

Table 1
Summary of the analysis: Techniques and their properties.

hard to build from scratch, but easier after the approximation achieved in the systematic approaches.

The requirement of defining all constraints explicitly is also a hard limitation of mixed integer programming. The amount of memory required, as well as the preprocessing step to generate them are also an issue that should not be forgotten, especially when dealing with large-scale problems. For example, in (Lim et al., 2005), heuristic approaches performed better than CPLEX in large-scale problems. In addition, if we wish to contemplate other issues such as delays or exceptions in our problem (see the complete definition of the problem in (López, 2005)), it is not so easy to imagine how these new constraints could be linearized. Thus, constructive models (like chronological backtracking, branch&bound, etc.) can manage these kinds of complex constraints. Moreover, the great importance of an adequate modeling in GRASP, Tabu and genetic algorithms approaches should be taken into account, as has been shown in the respective sections.

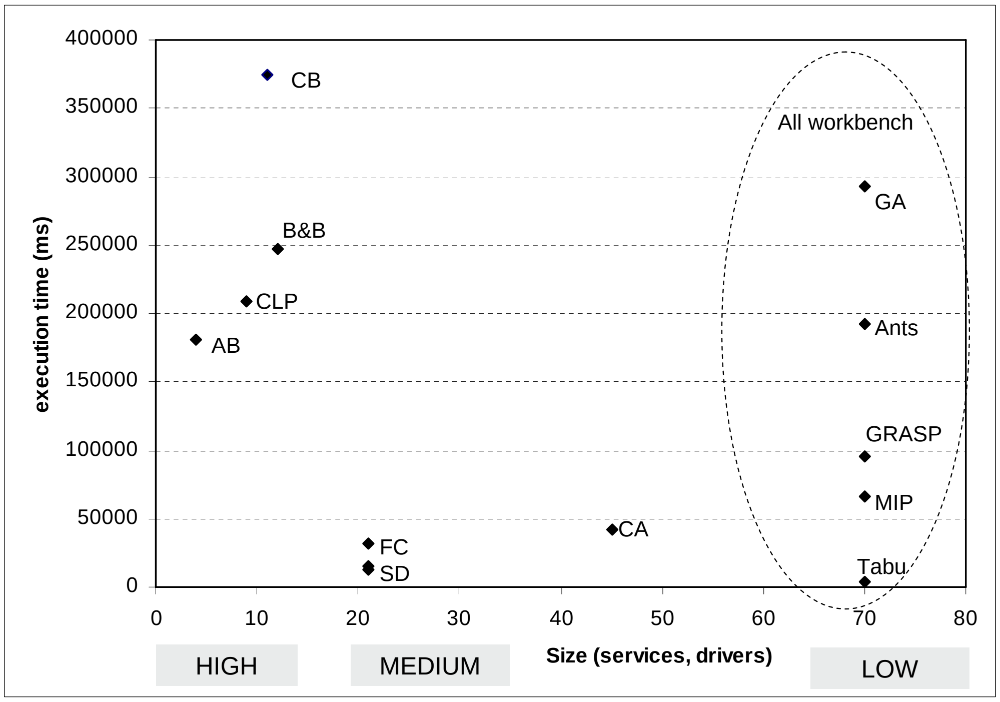

Fig. 7. Maximum problems managed by the methods and the associated execution time.

Conversely, distributed approaches and, particularly, the synchronous one that allows partitioning the problem in different sizes shows an average computational time has additional interesting features to continue their exploration. First of all, the synchronous distributed approach can take advantage of classical techniques when solving subproblems. In our experimental analysis, each subproblem was solved using Forward Checking methods. In addition, they exhibit good modeling expressiveness. Since it is backtracking based, it is possible to imagine the extension of the algorithm to exhibit an anytime behavior.

Regarding hybridization, it is also present in another of the techniques analyzed: combinatorial auctions. We have used mixed integer programming techniques to solve the winner determination problem that the combinatorial auction poses. The best current algorithm, CABOB, also uses MIP techniques (CPLEX) to perform estimations in order to prune the search space. Hybridization tries to use the advantages of classical methods, and it is a good direction to look in, since looking at Table 1 no single method can be claimed to be the best in all the properties.

Finally, we should remark that in this paper, only general methods and techniques for solving combinatorial problems have been analyzed. Some of these techniques make use of domain-independent heuristics in the search (for instance, constraints satisfaction techniques). Others methods (for instance,

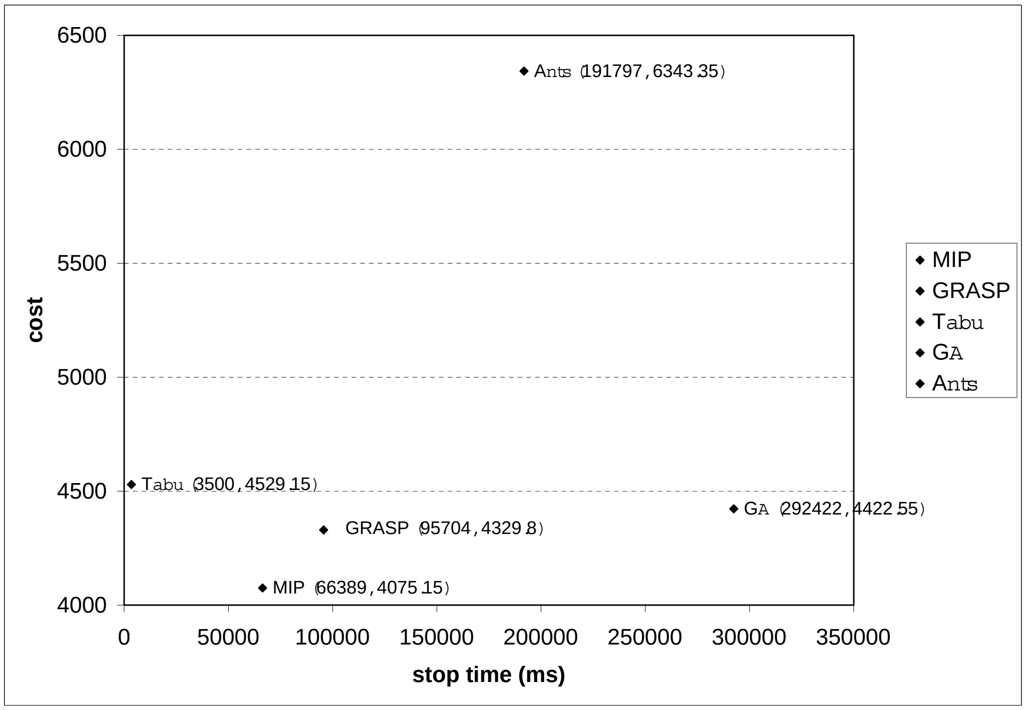

Fig. 8. Time (sec) required to find a solution for the 70 case and the cost of the solution. Only MIP finds the optimal solution.

branch&bound techniques), require the design of domain-dependent heuristics. However, we do not have deepened on the design of specific domain-dependent heuristics. It is clear that adequate domain-dependent heuristics, specifically designed for each kind of problems, can obtain good solutions in good computational times. But they would require a deeper work on each of the techniques and the results would strongly depend of each domain of problems. Therefore, the scope of our results would be too narrow to be of general interest. Our goal in this first approach to our problem is to show how general techniques can be applied to combinatorial problems, and the specific features of each approach in order to facilitate the selection of the appropriate technique to dedicate much more effort in solving the complete problem with the most suitable one.

## 7 Conclusions

When facing a new complex problem, especially in optimization, several techniques are available, either from the Operational Research or Artificial Intelligence fields. The selection of one technique or another is a critical issue. Even though in the literature there are some previous similar problems, slight differences in the problem data can decide upon the suitability of one technique or another. Experimentation with the problem data is often the only way to know, step by step, what the particular challenge of the problem is and what the different techniques offer. This is what we have done when trying to select the most appropriate technique for the road passenger transportation problem.

In this paper we have experimentally analyzed several techniques, from classical ones like search, constraint propagation and mixed integer programming, to new ones as metaheuristics, combinatorial auctions and distributed approach. All the results were obtained in a comparable PC configuration (Pentium IV 3GHz, 1Gb of RAM).

The results have shown us that even though mixed integer programming is the one that outperforms the best in a non large-scale problem, the modeling of a problem in this method is not always straightforward. A change in the problem data could derive in the revision of the complete model of the problem. Conversely, general search methods are more flexible and versatile and allow any new constraint to be defined at your convenience, but at a higher computational cost. New search distributed methods can offer a new opportunity for general search methods in order to improve efficiency, in addition to guaranteeing privacy and security issues on the data. This latter feature is interesting in our problem, since driving preferences can be added to the problem in a distributed way. In addition, distributed approaches offer us the possibility of hybridizing the solution, by merging classical well proven techniques in the subproblems.

To summarize our results, we provide a feature table, in which the main differences between the analyzed methods have been shown, as some guidelines that should help the choice of one technique or another, given a specific problem. We hope that this information can be useful helps to other engineers and researchers to understand the kind of optimization techniques currently available on the shell.

## Acknowledgements

This research work has been partially funded by the Spanish MEC projects TIN2004-06354-C02-02 and TIN2004-06354-C02-01, and by the DURSI Automation Engineering and Distributed Systems Group, 00296.

## References

Abbink, E., van t Wout, J., and Huisman, D. (2007). Solving Large Scale Crew Scheduling Problems by using Iterative Partitioning. ATMOS 2007, pp.96-106.

Alba E., Laguna M., and Marti R. (2003). Metodos Evolutivos. Ingenieria UC 10, 80-89.

Ali, S., Koeing, S., and Tambe, M. (2005). Preprocessing techniques for accelerating the dcop algoritm adopt. *In: AAMAS.*

Apt, K. (2003). *Principles of Constraint Programming.* Cambridge University Press.

Barták, R. (1999). Constraint programming: In pursuit of the Holy Grail. In *Proceedings of the Week of Doctoral Students (WDS), Prague, Czech Republic.*

Beasley, J. and Chu, P. (1996). A genetic algorithm for the set covering problem. *European Journal of Operational Research*, (94):392–404.

Cooper, M. and K. Farhangian, K. (1985). Multicriteria optimization for nonlinear integer-variable programming. *Large Scale Systems*, 9.

Cramton, P., Shoham, Y., and Steinberg, R. (2006). *Combinatorial Auctions.* The MiT Press.

Dechter, R. (2003). *Constraint Processing.* Morgan Kaufmann Publishers.

Dorigo, M., Caro, G. D., and Gambardella, L. Ant algorithms for discrete optimization. Artificial Life Spring 1999, Vol. 5, No. 2: 137-172.

Dorigo, M. and Di Caro, G. (1999). The ant colony optimization metaheuristic. In Corne, D., Dorigo, M., and Glover, F., editors, *New Ideas in Optimization*, pages 11–32. McGraw-Hill, London.

Feo, T. and Resende, M. (1995). Greedy randomized adaptive search procedures. *J. of Global Optimization*, (6):109–133.

Fernández, A.J., and Hill, P.M. (2000). A Comparative Study of Eight Constraint Programming LanguagesOver the Boolean and Finite Domains. *Constraints*, 5(3):275–301.

Garcia de la Banda, M., Hermenegildo, M., M., B., Dumortier, V., Janssens, G., and Simoens, W. (1996). Global analysis of constraint logic programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–614.

Garey, M.R. and Johnson, D.S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman.

Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 13(5):533–549.

Glover, F. and Kochenberger, G. e. (2003). *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management.* Springer.

Glover, F. and Laguna, M. (1997). *Tabu Search.* Kluwer Academic Publishers.

Hentenryck, P. V. (1989). *Constraint Satisfaction in Logic Programming.* The MIT Press.

Hirayama, K. and Yokoo, M. (1997). Distributed partial constraint satisfaction problem. In *Principles and Practice of Constraint Programming*, page 222.

Hoffman, K. and Padberg, M. (1996). Combinatorial and integer optimization. *Encyclopedia of Operations Research.*

Jaffar, J. and Maher, M. J. (1994). Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581.

Kalagnanam, J. and Parkes, D. (2004). *Auctions, Bidding and Exchange*, chapter 1. Kluwer.

Kohl, N. (2003). Solving the worlds largest crew scheduling problem. ORbit, pp. 8-12.

Laplagne, I., Raymond, S.K., and Kwan, A.S.K. (2005). A hybridised integer programming and local search method for robust train driver schedules planning. E. Burke and M. TRick (Eds.): PATAT 2004, LNCS 3616, pp. 71-85, Springer.

Leyton-Brown, K. (2003). *Resource Allocation in Competitive Multiagent Systems*. PhD thesis, Stanford University.

Lim, A., Rodrigues, B., and Zhu, Y. (2005). Airport gate scheduling with time windows. *Artificial Intelligence Review*, 24:5–31.

López, B. (2005). Time control in road passenger transportation. problem description and formalization. Research Report IIiA 05-04, University of Girona.

Maniezzo, V., Gambardella, L.M. and De Luigi, F. (2004). Ant Colony Optimization. Onwubolu, G. C. and Babu, B. V. (Eds): New Optimization Techniques in Engineering, Springer-Verlag Berlin Heidelberg, pp 101-117.

Maroto, C., Alcaraz, J., and Ruiz, R. (2003). *Investigación Operativa. Models i tècniques d'optimització*. Universitat Politècnica de València.

Melián, B., Moreno, J., and Moreno, J. (2003). Metaheuristics a global view. *Revista Iberoamericana de Inteligencia Artificial*, (19):7–28.

Modi, P., Shen, W., Tambe, M., and Yokoo, M. (2005). Adopt: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence Journal*, 161:149–180.

Orden, A. (1993). Lp from the '40s to the '90s. *Interfaces*, 23(5):2–12.

Portugal, R., Ramalhinho Lourenço, H., and Paixao, J.P. (2006). Driver Scheduling Problem Modelling. Working Papers (Universitat Pompeu Fabra. Departamento de Economa y Empresa), num. 991.

Ramalhinho Lourenço, H., Pinto Paixao, J., and Portugal, R. (2001). Metaheuristics for the bus-driver scheduling problem. *Transportation Science*, 3(35):331–343.

Rossi, F., van Beek, P., Walsh, T. (2006). Handbook of Constraint Programming. Elsevier.

Salido, M.A. and Barber, F. (2006). Distributed CSPs by Graph Partitioning. *Applied Mathematics and Computation*, (183), pages 491-498.

Sandholm, T. (2002). Algorithm for optimal winner determination in combinatorial auctions. *Artificial Intelligence*, (135):1.

Saxena, R.R. (2007). Enumeration Technique for Set Covering, Partitioning and Packing Problems with Non-Linear Objective Function: A Combinatorial Approach. ICCOPT II & MOPTA-07.

Yokoo, M., Durfee, E., Ishida, T., and Kuwabara, K. (1998). The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Trans. on Knowledge and DATA Engineering*, 10(5).

Yokoo, M. and Hirayama, K. (2000). Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3:185–207.

**Beatriz López** graduated in Computer Science from the Autonomous University of Barcelona in 1986 and received the PhD in Computer Science from the Technical University of Catalonia in 1993. Currently, she is lecturer and researcher at the Department of Electronics, Electricity and Automation Engineering at the University of Girona, Spain. Dra. López has published over 80 papers, and is editor of two books. Her research interests include optimization in distributed environments, marked-based optimization mechanisms, planning and scheduling, and case-based reasoning.

**Victor Muñoz** is currently a PhD student in Department of Electronics, Computer Engineering and Automatics at University of Girona. His current research includes optimization, robustness, constraint satisfaction problems, combinatorial auctions and complexity analysis.

**Javier Murillo** is currently a PhD student in Department of Electronics, Computer Engineering and Automatics at University of Girona. His current research include optimization, constraint satisfaction problems, combinatorial auctions, multi-agent systems and planning and scheduling.

**Federico Barber** is Full Professor with the Department of Computer Science, where he leads a research team in artificial intelligence. He has worked on the development of temporal reasoning systems, Constraint Satisfaction Problems, planning and scheduling. He is the author of about 90 research articles which have been published in several journals and conferences. His research has produced several tools for solving real-world optimization combinatory problems. He has participated in and led several national and European research projects related to these areas. He is currently president of the Spanish Association for Artificial Intelligence, and member of several scientific associations.

**Miguel A. Salido** is an associate professor in Computer Science at the Technical University of Valencia, Spain. Most of his research is focused on techniques for constraint satisfaction techniques and railway scheduling problems. He is the recipient of some national and international awards. He is author of more than 70 papers published on international journals and conferences. He is editor of some books and guess editor of some international journals. He is member of several Scientific and Organizing Committees.

**Montserrat Abril** has an honors degree in Computer Science. She is currently a PhD student in the Department of Information Systems and Computation at the Technical University of Valencia. She has a industrial background with the National Network of Spanish Railways. Recently, her work has diversified to include the development and study of robustness and railway capacity. His current research interests include metaheuristic methods, constrained optimization and distributed techniques for constraint satisfaction problems.

**Mariamar Cervantes** is currently a PhD student in the Department of Computer Science at the Technical University of Valencia. Her current research interests include metaheuristic methods, constrained optimization, and project scheduling.

**Luis Fernando Caro** received the B.S.E.E. degree from *Universidad del Norte*, in 2005. Currently, he is a Ph.D. student in the Broadband Communications and Distributed Systems group at the University of Girona. His research interest includes Carrier ethernet technologies, network optimization and GMPLS.

**Mateu Villaret** graduated in Computer Science from the Technical University of Catalonia in 1996 and in 2004 he obtained his Ph.D. from Technical University of Catalonia. He is currently a lecturer and researcher at the University of Girona. His main research is related with automatic theorem proving foundations, mainly in unification theory. He is currently working also in constraint solving problems via Sat Modulo Theories. He has several papers published in major conferences of the area like RTA, CADE, LPAR and IJCAR.