A Hybrid Approach for Solving Large Scale Crew Scheduling Problems

Tallys H. Yunes* tallys@acm.org

Arnaldo V. Moura arnaldo@dcc.unicamp.br

Cid C. de Souza† cid@dcc.unicamp.br

Institute of Computing, University of Campinas, Brazil

Abstract

We consider several strategies for computing optimal solutions to large scale crew scheduling problems, which are known to be notoriously difficult combinatorial optimization problems. Provably optimal solutions for very large real instances of such problems were computed using a hybrid approach that integrates mathematical and constraint programming techniques. A declarative programming environment was used to develop the constraint based models. The declarative nature of such an environment proved instrumental when modeling complex problem restrictions and, particularly, in efficiently searching the very large space of feasible solutions. The code was tested on real problem instances, stemming from the operation of two bus lines of a typical Brazilian transit company that serves a major urban area. Some of those instances contained an excess of 1.8×10^9 entries and could be solved to optimality in an acceptable running time when executing on a typical desktop PC.

Keywords: declarative programming, constraint programming, crew scheduling, hybrid algorithms, mathematical programming, column generation.

1 Introduction

Urban transit crew scheduling problems have been receiving a great deal of attention for the past decades. Since employee wages may well rise to 50 percent or more of a company's total expenditures, even small percentage savings over this cost can be quite significant. In this article, we report on a hybrid strategy that is capable of efficiently obtaining provably optimal solutions for some large instances of specific crew scheduling problems. These instances stem from the operational environment of a typical Brazilian transit company that serves a major urban area. The hybrid approach we developed meshes some classical Integer Programming (IP) techniques and some Constraint Satisfaction (CSP) techniques. This is done in such a way as to extract the power of these two approaches where they contributed their best towards solving the large scheduling problem instances considered.

The resulting code compiles under the Linux operating system, kernel 2.0. Running on a typical 350 MHz desktop PC with 320 MB of main memory, it computed optimal solutions for problem instances with an excess of 1.8×10^9 entries, in a reasonable amount of time.

*Supported by FAPESP grant 98/05999-4, and CAPES

†Supported by FINEP (ProNEx 107/97), and CNPq (300883/94-3)

In the sequel, we offer some general comments on the specific methods and techniques that were used. We started on a pure IP track applying a classical branch-and-bound technique to solve a set partitioning model. Since this method requires that all feasible duties are previously inserted into the problem formulation, all memory resources were rapidly consumed when we reached half a million feasible duties. To circumvent this difficulty, we implemented a column generation technique. As suggested in [5], the subproblem of generating feasible duties with negative reduced cost was transformed into a constrained shortest path problem over a directed acyclic graph and then solved with Dynamic Programming. However, due to the size and idiosyncrasies of our problem instances, this technique did not make progress towards solving larger instances.

In parallel, we also implemented a heuristic algorithm that had obtained very good results on large set covering problems [2]. With this implementation, problems with up to two million feasible duties could be solved to optimality. But this particular heuristic also requires that all feasible duties be present in memory during execution. Although some progress in time efficiency was achieved, memory usage was still a formidable obstacle.

The difficulties faced when modeling in the previous approaches almost disappeared when we turned to a declarative language that supports constraint specification over finite domain variables. We were able to implement our models in little time, producing code that was both concise and clear. When executed, it came as no surprise that the model showed two distinct behaviors, mainly due to the huge size of the search space involved. It was very fast when asked to compute new feasible duties, but lagged behind the IP methods when asked to obtain a provably optimal schedule. The search space generated by our problem instances is enormous and there are no strong local constraints available to help the resolution process and, also, a good heuristic to improve the search strategy does not come easily [4].

In order to harness the capabilities of both the IP and CSP techniques, we resorted to a hybrid approach to solve the larger, more realistic, problem instances. The main idea is to use the linear relaxation of a smaller core problem in order to efficiently compute good lower bounds on the optimal solution value. Using the values of dual variables in the solution of the linear relaxation, we can enter the generation phase that computes new feasible duties. This phase is declaratively modeled as a constraint satisfaction problem and submitted to the constraint solver. The solver returns new feasible duties to be inserted in the IP problem formulation, and the initial phase can be taken again, restarting the cycle. This approach secures the strengths of both the pure IP and the pure CSP formulations: only a small subset of all the feasible duties is efficiently dealt with at a time, and new feasible duties are quickly computed only when they will make a difference. The resulting code was tested on some large instances, based on real data. As of this writing, we can solve, in a reasonable time and with proven optimality, instances with an excess of 150 trips and 12 million feasible duties.

This article is organized as follows. Section 2 describes the crew scheduling problem. In Section 3, we discuss the IP approach and give implementation details and computational results. We also report on the implementation of two alternative techniques: standard column generation and heuristics. In Section 4, we investigate the CSP approach and we also discuss some implementation details and computational results. In Section 5, we present the hybrid approach. Implementation details are discussed and some computational results on real data are reported. Finally, in Section 6, we conclude and discuss further issues.

In the sequel, execution times inferior to one minute are reported as *ss.cc*, where *ss* denotes seconds and *cc* denotes hundredths of seconds. When reporting execution times that exceed 60 seconds, we use the alternative notation *hh*:*mm*:*ss*, where *hh*, *mm* and *ss* represent hours, minutes and seconds, respectively.

2 The Crew Scheduling Problem

In a typical crew scheduling problem, a set of trips have to be assigned to some available crew members. The goal is to assign a subset of the trips to each crew member in such a way that no trip is left unassigned. As usual, not every possible assignment is allowed since a number of constraints must be observed. Additionally, a cost function has to be minimized.

Among the following terms, some are of general use, while others reflect specifics of the transportation service for the urban area where the input data came from. A *depot* is a location where crews may change buses and rest. The act of driving a bus from one depot to another depot, passing by no intermediate depot, is named a *trip*. Associated with a trip we have its *start time*, its *duration*, its *initial depot*, and its *final depot*. The duration of a trip is statistically calculated from field collected data, and depends on many factors, such as the day of the week and the start time of the trip along the day. A *duty* is a sequence of trips that are assigned to the same crew. The *idle time* is the time interval between two consecutive trips in a duty. Whenever this idle time exceeds *Idle_Limit* minutes, it is called a *long rest*. During a long rest, crews can leave the premises and return later to resume their shift. A duty that contains a long rest is called a *two-shift duty*. The *rest time* of a duty is the sum of its idle times, not counting long rests. The parameter *Min_Rest* gives the minimum amount of rest time, in minutes, that each crew is entitled to. The sum of the durations of the trips in a duty is called its *working time*. The sum of the *working time* and the *rest time* gives the *total working time* of a duty. The time, in minutes, that a crew member works in excess of *Workday* minutes is called *over time* and is given by $\max\{0, \textit{total working time} - \textit{Workday}\}$. The *Workday* is a given parameter, specified by union regulations, that bounds the maximum time that an employee can work without incurring in over time. The *maximum over time* is an upper bound on *over time*. Finally, the *maximum working time* is given by $\textit{Workday} + \textit{maximum over time}$.

The input data comes in the form of a two dimensional table where each row represents one trip. For each trip, the table lists four columns with information about this trip: *start time*, measured in minutes after midnight, *duration*, measured in minutes, *initial depot* and *final depot*. We have used data that reflect the operational environment of two bus lines, Line 2222 and Line 3803, that serve the metropolitan area around the city of Belo Horizonte, in central Brazil. Line 2222 has 125 trips and one depot and Line 3803 has 246 trips and two depots. The input data tables for these lines are called OS 2222 and OS 3803, respectively. By considering initial segments taken from these two tables, we derived several other smaller problem instances. For example, taking the first 30 trips of OS 2222 gave us a new 30-trip problem instance. Table 1(a) shows the first 10 rows of OS 3803. A measure of the number of active trips along a typical day, for both Line 2222 and Line 3803, is shown in Table 1(b). This figure was constructed as follows. For each (x, y) entry, we consider a time window $T = [x, x + \textit{Workday}]$. The ordinate y indicates how many trips there are with start time s

(a)

Start	Dur	I. dep.	F. dep.
1	38	1	2
50	40	2	1
90	38	1	2
130	38	2	1
170	38	1	2
210	38	2	1
250	39	1	2
290	38	2	1
285	45	1	2
335	45	2	1

(b)

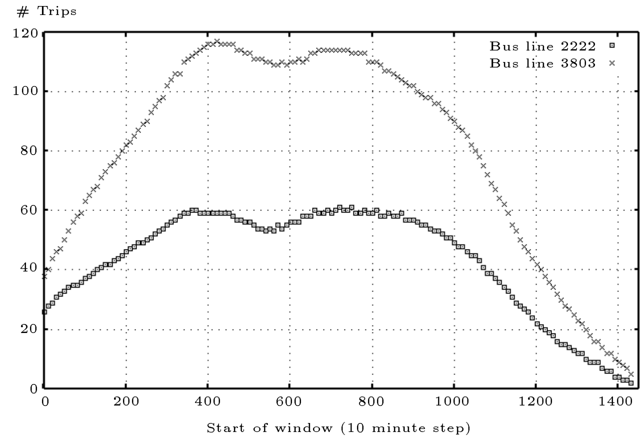

Table 1: (a) Sample from OS 3803. (b) Distribution of trips along the day.

and duration d such that $s \in T$ or $s + d \in T$.

For a duty to be classified as feasible, it has to satisfy many constraints imposed by labor contracts and union regulations, among others. The most important constraints are:

i. For each pair of consecutive trips, i and j, in every duty:
   (i) $(\textit{start time})_i + (\textit{duration})_i \leq (\textit{start time})_j$
   (ii) $(\textit{final depot})_i = (\textit{initial depot})_j$

ii. *total working time* $\leq$ *maximum working time*;

iii. *rest time* $+ \max\{0, \textit{Workday} - \textit{total working time}\} \geq$ *Min_Rest*;

iv. A duty can have at most one long rest interval;

For practical reasons, we used *maximum over time* $= 0$, *Idle_Limit* $= 120$, *Workday* $= 440$, and *Min_Rest* $= 30$, measured in minutes, A *feasible duty* is a duty that satisfies all problem constraints. A *schedule* is a set of feasible duties and an *acceptable schedule* is any schedule that partitions the set of all trips. The cost of a schedule is the sum of the costs of all its duties. Since the problem specification treats all duties as indistinguishable, every duty is assigned a unit cost. Hence, minimizing the cost of a schedule is the same as minimizing the number of crew members involved in the solution or, equivalently, the number of duties it contains. A *minimal schedule* is any acceptable schedule whose cost is minimal.

3 Mathematical Programming Approaches

Let m be the number of trips and n be the total number of feasible duties. The pure IP formulation of the problem is:

$$\min \sum_{j=1}^{n} c_j x_j \tag{1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} a_{ij} x_j = 1, \quad i = 1, 2, \ldots, m \tag{2}$$

$$x_j \in \{0, 1\}, \quad j = 1, 2, \ldots, n \tag{3}$$

The x_j's are 0-1 decision variables that indicate which duties belong to the solution and c_j is the cost of duty j. The coefficient a_{ij} equals 1 if duty j contains trip i, otherwise, a_{ij} is 0. This is a classical set partitioning problem where the rows represent all trips and the columns represent all feasible duties. Due to our simplifying assumption about the costs c_j, Equation (1) reduces to $\min \sum_{j=1}^{n} x_j$.

The main problem with the IP approach is clear: the number of feasible duties is enormous. We developed a declarative constraint program to count all feasible duties both in OS 2222 and in OS 3803. Table 2 summarizes the results for increasing initial sections of the input data tables. The time needed to count the number of feasible duties is also presented. When we consider all trips in OS 2222, we get in excess of one million feasible duties. Considering all trips in OS 3803, we reached more than 122 million feasible duties.

OS 2222 (1 depot)			**OS 3803 (2 depots)**		
# Trips	# Feas. Duties	Time	# Trips	# Feas. Duties	Time
10	63	0.07	20	978	1.40
20	306	0.33	40	6,705	5.98
30	1,032	0.99	60	45,236	33.19
40	5,191	5.38	80	256,910	00:03:19
50	18,721	21.84	100	1,180,856	00:18:34
60	42,965	00:01:09	120	3,225,072	00:57:53
70	104,771	00:03:10	140	8,082,482	02:59:17
80	212,442	00:05:40	160	18,632,680	08:12:28
90	335,265	00:07:48	180	33,966,710	14:39:21
100	496,970	00:10:49	200	54,365,975	17:55:26
110	706,519	00:14:54	220	83,753,429	42:14:35
125	1,067,406	01:00:27	246	122,775,538	95:49:54

Table 2: Number of feasible duties for OS 2222 and OS 3803.

3.1 Pure IP Approach

In the pure IP approach, we used the declarative constraint program to generate an output file containing all feasible duties. A C program was developed to convert the output file to conform with the CPLEX[1] input format. The resulting file was fed into a CPLEX 3.0 LP solver. The node selection strategy used was *best-first* and branching was done upon the most fractional variable. Every other setting of the branch-and-bound algorithm used the standard default CLPEX configuration.

Computational results for OS 2222 appear in Table 3, columns under "Pure IP". It soon became apparent that the pure IP approach using the CPLEX solver would not be capable of obtaining the optimal solution for the smaller OS 2222 problem instance. Besides, memory usage was also increasing at an alarming pace, and execution time was lagging behind when compared to other approaches that were being developed in parallel. As an alternative, we decided to implement a column generation approach.

[1]CPLEX is a registered trademark of ILOG, Inc.

3.2 Column Generation with Dynamic Programming

Column generation is a technique that is widely used to handle linear programs which have a very large number of columns in the coefficient matrix. The method works by repeatedly executing two phases. In a first phase, instead of solving a linear relaxation of the whole problem, in which all columns are required to be loaded in memory, we quickly solve a smaller problem, called the *master* problem, that deals only with a subset of the original columns. That smaller problem solved, we start phase two, looking for columns with a negative reduced cost. If there are no such columns, we have proved that the solution at hand indeed minimizes the objective function. Otherwise, we augment the master problem by bringing in a number of columns with negative reduced cost, and start over on phase one. The problem of computing columns with negative reduced costs is called the *slave* subproblem. When the original variables have integer values, this algorithm must be embedded in a branch-and-bound strategy. The resulting algorithm is also known as *branch-and-price*.

In order to start the algorithm, one has to decide how to setup the first master problem. As a general guideline, one should avoid not only to initialize the master problem with trivial columns, e.g. the identity matrix, but also some apparently good collection of columns that may cause the method to wander into unpromising regions [13]. In our case, however, a trivial initialization worked best. In an attempt to get a better performance, we augmented the initial identity matrix with a set of columns computed using the declarative constraint program discussed in Section 4. Computational results did not favor this alternative and we refrained from using it in subsequent experiments.

# Trips	# Feasible Duties	Opt	**Pure IP**		**CG+DP**		**Heuristic**	
			Sol	Time	Sol	Time	Sol	Time
10	63	7	7	0.02	7	0.01	7	0.05
20	306	11	11	0.03	11	0.07	11	0.30
30	1,032	14	14	0.06	14	0.52	14	10.37
40	5,191	14	14	3.04	14	9.10	14	13.02
50	18,721	14	14	14.29	14	00:01:29	14	00:30:00
60	42,965	14	14	00:01:37	14	00:07:54	14	00:30:22
70	104,771	14	14	00:04:12	14	00:44:19	14	00:03:28
80	212,442	16	16	00:33:52	16	03:53:58	16	00:16:24
90	335,265	18	18	00:50:28	18	08:18:53	18	00:22:42
100	496,970	20	20	02:06:32	20	15:08:55	20	00:50:01
110	706,519	22	-	-	-	-	22	01:06:17
125	1,067,406	25	-	-	-	-	25	01:55:12

Table 3: Computational results for OS 2222 (1 depot).

The master problems were solved using the CPLEX 3.0 LP solver.

In general, the slave subproblem can also be formulated as another IP problem. In our case, constraints like the one on two-shift duties substantially complicate the formulation of a pure IP model. As another approach, Desrochers and Soumis [5] suggest reducing the slave problem to a constrained shortest path problem, formulated over a related directed acyclic graph G. When the algorithm for solving the slave problem is about to start, the value u_i of all the dual variables can easily be extracted from the linear relaxation solution of the current master problem. We include in G a *trip arc* to represent each trip i and assign to it a cost $-u_i$. An arc with cost zero connects the end vertex of a trip arc i to the initial vertex of a trip arc j whenever trips i and j do not overlap in time. Also, zero cost arcs connect a source node s to the initial vertices of all trip arcs, and some other zero cost arcs connect the end vertex of all trip arcs to a sink node t. A path p from s to t in G represents a duty D, and the cost associated to p is $-\sum_{i\in D} u_i$. From the IP formulation, we know that the reduced cost of a duty D is given by the expression $1-\sum_{i\in D} u_i$. Hence, to obtain a duty with negative reduced cost we seek a path in G whose associated cost is smaller than -1. But, we also need to guarantee that such a path represents a feasible duty. To this end, the algorithm keeps track of the resource consumption of each path it is dynamically constructing. When the next trip arc is added to a path, the latter becomes infeasible if this trip arc depletes any resource beyond its limits. This process terminates when the sink node is reached. Besides the cost, we used three resources representing the total working time, the total rest time and a binary value that indicated if the path is a two-shift duty. Since different paths consume resources in different amounts, the implementation is further complicated because it is necessary to maintain, at each node, a list of feasible paths that can reach that node. A path that reaches a node can only be discarded if it consumes more of all resources than another path that also reaches that same node. As a bonus, when the algorithm terminates, it is easy to extract not only the shortest feasible path, but also a number of additional feasible paths, all with negative reduced costs. We used these ideas, complemented by other observations from Beasley and Christofides [1] and our own experience.

To implement the branch-and-price strategy, the use of the ABACUS[2] branch-and-price framework (version 2.2) saved a lot of programming time. One of the important issues was the choice of the branching rule. When applying a branch-and-bound algorithm to set partitioning problems, a simple branching rule is to choose a binary variable and set it to 1 on one branch and set it to 0 on the other branch, although there are situations where this might not be the best choice [13]. Ryan and Foster [12] suggest an improved branching rule. However, our computational results using the simpler branching rule had already shown a very small number of nodes in the implicit enumeration tree (less than 42). We judged that any possible marginal gains did not justify, in our case, the extra programming effort required to implement a more elaborated branching rule.

Computational results for OS 2222 appear in Table 3, columns under “CG+DP”. This approach did not reach a satisfactory time performance, mainly because the constrained shortest path subproblem is relatively loose. Being a pseudo-polynomial algorithm, the state space at each node of the graph has the potential of growing exponentially with the input size. The number of feasible paths the algorithm has to maintain became so large that the time spent looking for columns with negative reduced cost is responsible for more than 97% of the total execution time, on the average.

3.3 A Heuristic Approach

Heuristics offer another approach to solve scheduling problems and there are many possible variations. Initially, we set aside those heuristics that were unable to reach an optimal

[2] `http://www.informatik.uni-koeln.de/ls_juenger/projects/abacus.html`

solution. As a promising alternative, we decided to implement the set covering heuristic developed by Caprara et al. [2]. This heuristic won the FASTER competition jointly organized by the Italian Railway Company and AIRO, solving, in reasonable time, large set covering problems arising from crew scheduling. Using our own experience and additional ideas from the chapter on Lagrangian Relaxation in [11], an implementation was written in C and went through a long period of testing and benchmarking. Tests executed on set covering instances coming from the OR-Library showed that our implementation is competitive with the original CFT implementation in terms of solution quality. When this algorithm terminates, it also produces a lower bound for the optimal covering solution, which also could be used as a bound for the partition problem. We verified, however, that on the larger instances, the solution produced by the heuristic turned out to be a partition.

# Trips	# Feasible Duties	Opt	**Heuristic**	
			Sol	Time
20	978	6	6	0.35
40	6,705	13	13	3.60
60	45,236	15	15	52.01
80	256,910	15	15	00:08:11
100	1,180,856	15	15	00:13:51
110	2,015,334	15	15	00:23:24

Table 4: Heuristic over OS 3803 (2 depots).

Computational results for OS 2222 appear in Table 3, columns under "Heuristic". Comparing all three implementations, it is clear that the CFT heuristic gave the best results. However, applying this heuristic to the larger OS 3803 data set was problematic. Since storage space has to be allocated to all feasible columns, memory usage becomes prohibitive. It was possible to solve instances with up to 2 million feasible duties, as indicated in Table 4. Beyond that limit, 320 MB of main memory were not enough for the program to terminate.

4 Constraint Programming Approach

Modeling with finite domain constraints is rapidly gaining acceptance as a promising declarative programming environment to solve large combinatorial problems. This led us to also model the crew scheduling problem using pure declarative constraint programming techniques. All models described in this section were formulated using the ECL^{i}PS^e [3] syntax, version 4.0. Due to its large size, the ECL^{i}PS^e formulation for each run was obtained using a program generator that we developed in C.

A simple pure CP formulation for the scheduling problem was developed using a list of items, each item being itself a list describing an actual duty. A number of recursive predicates guarantee that each item satisfies all labor and regulation constraints (see Section 2), and also enforce restrictions of time and depot compatibility between consecutive trips. These feasibility predicates iterate over all list items. The declarative database contains one fact for each line of input data, as explained in Section 2. The resulting model is very simple to program in a declarative environment. The formulation, however, did not reach

[3] `http://www.icparc.ic.ac.uk/eclipse`

satisfactory results when submitted to the ECL^{i}PS^e solver, as shown in Table 5. A number of different labeling techniques, different clause orderings and several variants on constraint representation were explored, to no avail. When proving optimality, the situation was even worse. It was not possible to prove optimality for instances with only 10 trips in less than an hour of execution time. The main reason for this poor performance resides on the recursive nature of the list representation, and the absence of reasonable lower and upper bounds on the value of the optimal solution that could aid the solver discard unpromising labelings.

To improve the execution time, we abandoned the pure list representation and created a model in which it was easier to directly impose algebraic constraints. The new model is based on a two dimensional matrix X of integers. The number of columns in X is an upper bound on the size of any feasible duty (*UBdutyLen*), and the number of rows in X is an upper bound on the total number of duties needed to create the schedule (*UBnumDut*). To calculate *UBdutyLen*, we start by summing up the durations of the trips, taken in non-decreasing order. When we reach a value that is greater than *maximum working time* minutes, *UBdutyLen* is set to the number of trips added. Initially, we used the number of trips as a rough estimate for *UBnumDut*. As the definitive value for *UBnumDut* we used the number of duties on the first feasible solution found by the solver. Each X_{ij} element, called a *cell*, represents a single trip and is a finite domain variable with domain $[1..NT]$, where $NT = UBdutyLen \times UBnumDut$. Let N be the number of trips. It is clear that $N \leq NT$. Trips numbered $N+1$ to NT are *dummy trips*. The start time of the first dummy trip equals the arrival time of the last actual trip plus one minute and its duration is one minute. All the subsequent dummy trips also last one minute and their start times are such that no idle time between consecutive dummy trips exists, i.e., they start at each following minute. Their departure and arrival depots are equal to 0. With these choices, we prevent incompatibilities arising from time intersection and mismatching of depots among the dummy trips. Besides, we avoid the occurrence of a dummy trip between two real trips in a feasible duty.

Using this representation, the set partitioning condition can be easily achieved with an `alldifferent` constraint applied to a list that contains all the cells. Also note that, the way *UBdutyLen* is calculated assures that at least one dummy trip appears in every line in X. Moreover, their start times guarantee that the dummy trips occupy consecutive cells at the end of every line. This is on purpose, to facilitate the representation of some constraints.

Five additional matrices were used: *Start*, *End*, *Dur*, *DepDepot* and *ArrDepot*. Cell (i, j) of these matrices represents, respectively, the start time, the end time, the duration, and the departure and arrival depots of trip X_{ij}. Next, we state constraints in the form `element(`$X_{ij}, S, Start_{ij}$`)`, where S is a list containing the start times of the first NT trips. The semantics of this constraint assures that $Start_{ij}$ is the k-th element of list S where k is the value in X_{ij}. This maintains the desired relationship between matrices X and *Start*. Whenever X_{ij} is updated, $Start_{ij}$ is also modified, and vice-versa. Similar constraints are stated between X and each one of the four other matrices. Now, we can use these new matrices to easily state additional constraints, like:

$$End_{ij} \leq Start_{i(j+1)} \tag{4}$$

$$ArrDepot_{ij} + DepDepot_{i(j+1)} \neq 3 \tag{5}$$

$$Idle_{ij} = Start_{i(j+1)} - End_{ij} \tag{6}$$

for all $i \in \{1, \ldots, \mathit{UBnumDut}\}$ and all $j \in \{1, \ldots, \mathit{UBdutyLen}-1\}$. Equation (4) guarantees that trips overlapping in time are not in the same duty. Since the maximum number of depots is two, an incompatibility of two consecutive trips occurs only when the ending depot is 1 and the starting depot is 2, or vice-versa. Equation (5) forbids that situation. Additionally, the consecutiveness of two dummy trips is permitted (for the sum of the depots equals 0) and the appearance of the first dummy trip after the last real trip in a duty is not precluded by this constraint, because the sum of the depots in this case can only assume the values 1 or 2. With a one depot instance, these constraints are not necessary and they are omitted, together with the *ArrDepot* and *DepDepot* matrices.

Some other constraints are expressed using the $Idle_{ij}$ variables of Equation (6). The constraint on total working time, for each duty i, is given by:

$$TWT_i = \Big(\sum_{j=1}^{UBdutyLen-1} BD_{ij} \times Dur_{ij}\Big) + \Big(\sum_{j=1}^{UBdutyLen-1} BI_{ij} \times Idle_{ij}\Big) \tag{7}$$

$$TWT_i \leq \textit{maximum working time} \tag{8}$$

Where BD_{ij} and BI_{ij} are binary variables such that $BD_{ij} = 1$ iff X_{ij} contains an actual trip, and $BI_{ij} = 1$ iff $Idle_{ij} \leq Idle\_Limit$. The constraint on total rest time is:

$$\Big(\sum_{j=1}^{UBdutyLen-1} Idle_{ij}\Big) + \max\{0, Workday - TWT_i\} \geq Min\_Rest \tag{9}$$

for each duty i. For two-shift duties, we impose that at most one of the $Idle_{ij}$ variables can assume a value greater than *Idle_Limit*.

The execution time of this model was further improved by:

Elimination of symmetries — Two different assignments to the X_{ij} variables that differ only by a permutation of the lines of the matrix represent the same final schedule. These equivalent solutions have been eliminated by imposing that the the first column of the X matrix is sorted in increasing order. Exchanging the position of dummy trips in a feasible solution gives another solution that is equivalent to the original one. New constraints were imposed over the X cells in order to force dummy trips to have only one possible placement in X, given that the actual trips had already been positioned.

Domain reduction — Certain trips can only appear on a subset of the available cells. For instance, the first real trip can only appear on cell $X_{1,1}$.

Use of another viewpoint — Different viewpoints [3] were also used. New Y_k variables were introduced representing "the cell that stores trip k", as opposed to the X_{ij} variables that mean "the trip that is put in cell ij" (an ij cell can be represented by the number $(i-1) \times \mathit{UBdutyLen} + j$). The new Y_k variables were then connected to the X_{ij} variables through *channeling constraints*. The result is a redundant model with improved propagation properties.

Different labeling strategies — Various labeling strategies have been tried, including the one developed by Jourdan [9]. The strategy of choosing the next variable to label as the one with the smallest domain (*first-fail* principle) was the most effective one. After choosing a variable, it is necessary to select a value from its domain following a specific order, when backtracking occurs. We tested different labeling orders, like increasing, decreasing, and also middle-out and its reverse. Experimentation showed that labeling by increasing order achieved the best results. On the other hand, when using two viewpoints the heuristic developed by Jourdan rendered the model approximately 15 % faster. The basic idea is to label a X variable according to the domain size of the associated Y variables. In our case, for instance, if the current domain of variable $X_{2,5}$ is $[1, 7, 8]$, the first value to be selected for labeling will be 8 if and only if Y_8 has the smallest domain among variables Y_1, Y_7 and Y_8.

The improved purely declarative model produced feasible schedules in a very good time, as indicated in Table 5. Obtaining provably optimal solutions, however, was still out of reach

# Trips	# Feasible Duties	First Model		Improved Model			
		Feasible		**Feasible**		**Optimal**	
		Sol	Time	Sol	Time	Sol	Time
10	63	7	0.35	7	0.19	7	0.63
20	306	11	12.21	11	0.47	11	9.22
30	1,032	15	00:02:32	15	0.87	14	00:29:17
40	5,191	15	00:14:27	15	0.88	-	> 40:00:00
50	18,721	15	00:53:59	15	0.97	-	-
60	42,965	-	-	15	2.92	-	-
70	104,771	-	-	16	3.77	-	-
80	212,442	-	-	19	8.66	-	-
90	335,265	-	-	24	17.97	-	-
100	496,970	-	-	27	29.94	-	-
110	706,519	-	-	27	39.80	-	-
120	952,620	-	-	28	00:01:07	-	-
125	1,067,406	-	-	32	00:01:21	-	-

Table 5: Pure CP models, OS 2222 data set.

for this model. Others have also reported difficulties when trying to solve crew scheduling problems with a pure declarative approach [4, 8]. Finding the optimal schedule reduces to choosing, from an extremely large set of elements, a minimal subset that satisfies all the problem constraints. The huge search space can only be dealt with satisfactorily when pruning is enforced by strong local constraints. Besides, a simple search strategy, lacking good problem specific heuristics, is very unlikely to succeed. When solving scheduling problems of this nature to optimality, none of the these requirements can be met easily, rendering it intrinsically difficult for pure CP techniques to produce satisfactory results in these cases.

5 A Hybrid Approach

Recent research [6] has shown that, in some cases, neither the pure IP nor the pure declarative CP approaches are capable of solving certain kinds of combinatorial problems satisfactorily. But a hybrid strategy may outperform these two methods.

When contemplating a hybrid strategy, it is necessary to to decide which part of the problem will be handled by a constraint solver, and which part will be dealt with in a classical way. Given the huge number of columns at hand, a column generation approach seemed to be almost mandatory. As reported in Section 3.2, we already knew that the dynamic programming column generator used in the pure IP approach did not perform well. On the other hand, a declarative language is particularly suited to express not only the constraints imposed by the original problem, but also the additional constraints that must be satisfied when looking for feasible duties with a negative reduced cost. Given that, it was a natural decision to implement a column generation approach where new columns were generated on demand by a declarative constraint program. Additionally, the discussion on Section 4 indicated that the declarative CP strategy implemented was very efficient when identifying feasible duties. It lagged behind only when computing a provably optimal solution to the original scheduling problem, due to the minimization constraint. Since it is not necessary to find a column with *the* most negative reduced cost, the behavior of the CP solver was deemed adequate. It remained to program the CP solver to find a set of new feasible duties with the extra requirement that their reduced cost should be negative.

The basis of this new algorithm is the same as the one developed for the column generation approach, described in Section 3.2. The dynamic programming routine is substituted for an ECL^{i}PS^e process that solves the slave subproblem and uses sockets to communicate the solution back to the ABACUS process. The initialization phase and the branching rule of the original algorithm remained unchanged. When the ABACUS process has solved the current master problem to optimality, it sends the values of the dual variables to the CP process. If there remain some columns with negative reduced costs, some of them are captured by the CP solver and sent back to the ABACUS process, and the cycle starts over. If there are no such columns, the LP solver has found an optimal solution. Having found the optimal solution for this node of the enumeration tree, its dual bound has also been determined. The normal branch-and-bound algorithm can then proceed until it is time to solve another LP at a different node of the enumeration tree.

The code for the CP column generator is almost identical to the code for the improved CP model, presented in Section 4. There are three major differences. Firstly, the matrix X now has only one row, since we are interested in finding *one* feasible duty and not a complete solution. Secondly, there is an additional constraint stating that the sum of the values of the dual variables associated with the trips in the duty being constructed should be greater than 1 (see Section 3.2). Finally, the minimization predicate was exchanged for a predicate that keeps on looking for new feasible duties until the desired number of feasible duties with negative reduced costs have been computed, or until there are no more feasible assignments. By experimenting with the data sets at hand, we determined that the number of columns with negative reduced cost to request at each iteration of the CP solver was best set to 53. Since minimization is not present any more, the redundant modeling, as well as the heuristic suggested by Jourdan, both used to improve the performance of the original CP formulation, now represented unnecessary overhead, and were removed.

The hybrid approach was able to construct an optimal solution to substantially larger instances of the problem, in a reasonable time. Computational results for OS 2222 and OS 3803 appear on Tables 6 and 7, respectively. Column headings have the following meanings: #Trips is the number of trips; #FD stands for the number of feasible duties; Opt is the value of the optimal integer solution; DBR is the dual bound at the root node of the

#Trips	#FD	Opt	DBR	#CA	#LP	#Nodes	PrT	LPT	TT
10	63	7	7	53	2	1	0.08	0.02	0.12
20	306	11	11	159	4	1	0.30	0.04	0.42
30	1,032	14	14	504	11	1	1.48	0.11	2.07
40	5,191	14	14	1,000	26	13	8.03	0.98	9.37
50	18,721	14	14	1,773	52	31	40.97	3.54	45.28
60	42,965	14	14	4,356	107	41	00:04:24	14.45	00:04:40
70	104,771	14	14	2,615	58	7	00:01:36	4.96	00:01:42
80	212,442	16	16	4,081	92	13	00:01:53	18.84	00:02:13
90	335,265	18	18	6,455	141	11	00:02:47	31.88	00:03:22
100	496,970	20	20	8,104	177	13	00:06:38	51.16	00:07:34
110	706,519	22	22	11,864	262	21	00:16:53	00:02:28	00:19:31
125	1,067,406	25	25	11,264	250	17	00:19:09	00:01:41	00:21:00

Table 6: Hybrid algorithm, OS 2222 data set (1 depot).

branch-and-bound enumeration tree; #CA is the number of columns added throughout each execution; #LP is the number of linear programming relaxations solved; and #Nodes is the number of tree nodes visited. The execution times are divided in three columns: PrT is the time spent generating columns; LPT is the time spent solving linear programming relaxations and TT is the total execution time. Note that, in every instance tested, the dual bound at the root node was equal to the value of the optimal integer solution. Hence, the LP relaxation of the problem already provided the best possible lower bound on the optimal solution value. Also note that the number of nodes visited by the algorithm was kept small. The same behavior can be observed with respect to the number of added columns.

The sizable gain in performance can be seen in the last three columns of each table. It is interesting to note, firstly, that the time taken to solve all linear relaxations of the problem was a small fraction of the total running time for both data sets.

It is also clear, from Table 6, that the hybrid approach was capable of constructing a provably optimal solution for the complete smaller data set using 21 minutes of running time on a typical 350 MHz desktop PC. That problem involved in excess of one million feasible columns and was solved considerably faster when compared with the heuristic approach, which was the best performer among all the previous approaches (see Section 3.3).

The structural difference between both data sets can be appreciated by observing the entries on the line that corresponds to 100 trips, in Table 7. The number of feasible duties on this line corresponds, approximately, to the same number of one million feasible duties that are present in the totality of 125 trips of the first data set, OS 2222. Yet, the algorithm used roughly twice as much time to construct the optimal solution when running over a test case that contained the first 100 trips of the larger data set, as it did when taking the 125 trips of the smaller data set. Also, the algorithm lagged behind the heuristic for OS 3803, although the latter was unable to go beyond 110 trips, due to excessive memory usage.

Finally, when we fixed a maximum running time of 24 hours, the algorithm was capable of constructing a solution, and prove its optimality, for as many as 150 trips taken from the larger data set. This corresponds to an excess of 12 million feasible duties. It is noteworthy that less than 60 MB of main memory were needed for this run to reach completion. A problem instance with as many as $150 \times (12.5 \times 10^6)$ entries would require over 1.8 GB

#Trips	#FD	Opt	DBR	#CA	#LP	#Nodes	PrT	LPT	TT
20	978	6	6	278	7	1	2.11	0.08	2.24
30	2,890	10	10	852	19	1	9.04	0.20	9.38
40	6,705	13	13	2,190	48	1	28.60	1.03	30.14
50	17,334	14	14	4,220	94	3	00:01:22	3.95	00:01:27
60	45,236	15	15	8,027	175	1	00:03:48	14.81	00:04:06
70	107,337	15	15	11,622	258	1	00:07:42	40.59	00:08:37
80	256,910	15	15	8,553	225	1	00:10:07	47.12	00:10:58
90	591,536	15	15	9,827	269	1	00:14:34	00:02:04	00:16:43
100	1,180,856	15	15	13,330	375	1	00:39:03	00:04:37	00:43:49
110	2,015,334	15	15	13,717	387	1	01:19:55	00:03:12	01:23:19
120	3,225,072	16	16	18,095	543	13	04:02:18	00:09:09	04:11:50
130	5,021,936	17	17	28,345	874	23	06:59:53	00:30:16	07:30:56
140	8,082,482	18	18	27,492	886	25	13:29:51	00:28:56	13:59:40
150	12,697,909	19	19	37,764	1,203	25	21:04:28	00:49:13	21:55:25

Table 7: Hybrid algorithm, OS 3803 data set (2 depots).

of main memory, if needed to be loaded into main memory. By efficiently dealing with only a small subset of the feasible duties, our algorithm managed to surpass the memory consumption bottleneck and could solve instances that were very large. This observation supports our view that a declarative CP formulation of column generation was the right approach to solve very large crew scheduling problems.

6 Conclusions and Future Work

Real world crew scheduling problems often give rise to large set covering or set partitioning formulations. We have shown a way to integrate pure Integer Programming and declarative Constraint Satisfaction Programming techniques in a hybrid column generation algorithm that solves, to optimality, huge instances of some real world crew scheduling problems. These problems appeared intractable for both approaches when taken in isolation. Our methodology combines the strengths of either side, getting over their main weaknesses and reaching a very successful blend.

Another crucial advantage of our hybrid approach over a number of previous attempts is that it considers *all* feasible duties. Therefore, the need does not arise to use specific rules to select, at the start, a subset of "good" feasible duties. This kind of preprocessing could prevent the optimal solution from being found. Instead, our algorithm implicitly looks at the set of all feasible duties, when activating the column generation method. When declarative constraint satisfaction formulations are applied to generate new feasible duties on demand, they have shown to be a very efficient strategy, in contrast to Dynamic Programming.

We believe that our CP formulations can be further improved. In particular, the search strategy deserves more attention. Earlier identification of unpromising branches in the search tree can reduce the number of backtracks and lead to substantial savings in computational time. Techniques such as dynamic backtracking [7] and the use of *nogoods* [10] can be applied to traverse the search tree more efficiently, thereby avoiding useless work.

We would like to thank the Pioneira Bus Company from the city of Belo Horizonte, in Brazil, for kindly providing us with the data on which our experiments have been based.

References

[1] J. E. Beasley and N. Christofides. An algorithm for the resource constrained shortest path problem. *Networks*, 19:379–394, 1989.

[2] A. Caprara, M. Fischetti, and P. Toth. A heuristic method for the set covering problem. Technical Report OR-95-8, DEIS, University of Bologna, Italy, 1995.

[3] B. M. W. Cheng, K. M. F. Choi, J. H. M. Lee, and J. C. K. Wu. Increasing constraint propagation by redundant modeling: an experience report. *Constraints*, 1998. Accepted for publication.

[4] K. Darby-Dowman and J. Little. Properties of some combinatorial optimization problems and their effect on the performance of integer programming and constraint logic programming. *INFORMS Journal on Computing*, 10(3), 1998.

[5] M. Desrochers and F. Soumis. A column generation approach to the urban transit crew scheduling problem. *Transportation Science*, 23(1), 1989.

[6] C. Gervet. Large Combinatorial Optimization Problems: a Methodology for Hybrid Models and Solutions. In *JFPLC*, 1998.

[7] M. L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, (1):25–46, 1993.

[8] N. Guerinik and M. Van Caneghem. Solving crew scheduling problems by constraint programming. In *Lecture Notes in Computer Science*, pages 481–498, 1995. Proceedings of the First International Conference on the Principles and Practice of Constraint Programming, CP'95.

[9] J. Jourdan. *Concurrent Constraint Multiple Models in CLP and CC Languages: Toward a Programming Methodology by Modeling.* PhD thesis, Denis Diderot University, Paris VII, February 1995.

[10] J. Lever, M. Wallace, and B. Richards. Constraint logic programming for scheduling and planning. *BT Technical Journal*, (13):73–81, 1995.

[11] C. R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems.* Wiley, 1993.

[12] D. M. Ryan and B. A. Foster. An integer programming approach to scheduling. In A. Wren, editor, *Computer Scheduling of Public Transport.* North-Holland Publishing Company, 1981.

[13] F. Vanderbeck. *Decomposition and Column Generation for Integer Programming.* PhD thesis, Université Catholique de Louvain, CORE, September 1994.