1 DISJUNCTIVE PROGRAMMING AND COOPERATING SOLVERS

Ken McAloon
Carol Tretkoff
Gerhard Wetzel

Logic Based Systems Lab
Brooklyn College and CUNY Graduate Center
Brooklyn, NY 11210
{tretkoff,mcaloon,gw}@sci.brooklyn.cuny.edu

Abstract:
There are two fundamental themes in constraint programming. One is discrete or finite domain constraint programming based on the constraint satisfaction model. The other is continuous constraint programming based on linear programming and its extensions. In this paper we propose techniques for making constraint solvers cooperate, and we present a scheduling application of the Dutch Railways and a new kind of algorithm for solving disjunctive programming problems, one which could not be developed without cooperating solvers. What emerges is that cooperating solvers, which have old roots in special purpose operations research methods, constitute a central technology with wide applicability.

1.1 INTRODUCTION

While discrete constraint satisfaction methods may be ideally suited to solve some problems (e.g. job-shop scheduling problems) and continuous methods to solve others (e.g. airline crew scheduling problems), there are also examples where each method on its own exhibits certain weaknesses, but where the problem can be addressed by the combination of both methods.

Work in this area has roots in CLP($\mathbb{R}$) [Jaffar and Michaylov 87] and Prolog III [Colmerauer 87] which brought linear programming (LP) into constraint logic programming. CHIP [Di*88] included unification over the Herbrand Universe, finite domain propagation for discrete variables, and a simplex solver for continuous variables; a fourth constraint solver took care of Boolean variables. To the best of our knowledge, the term *cooperating solver* was first used by DeBacker and Beringer [DeBacker and Beringer 95a] who obtained promising results using both the discrete and continuous solvers built into ILOG to solve multi-knapsack problems [DeBacker and Beringer 95b]. A team at Imperial College has linked the constraint logic programming system ECL^{i}PS^e with the MIP solver Cplex [Schimpf and Novello 97]. For a different many-sided view, see [Wetzel 97].

In this work we have used the 2LP system [McAloon and Tretkoff 97] to solve a multicommodity integer network flow problem of the Dutch Railways (*Nederlandse Spoorwegen*) which is greatly complicated by additional constraints on the coupling and decoupling of train carriages. A constraint-based language with C-like syntax, 2LP handles the continuous constraints. The discrete constraints are handled in auxiliary C code, which is called from the 2LP program. The C code and the 2LP code collaborate in the search process. For the example in this paper, the Cplex LP solver is used (transparently) as the LP engine.

The paper is organized as follows. In section 1.2 we present the background theory of disjunctive linear programming problems and the general concepts of *cuts* and of *tightenings*. Section 1.4 describes our sample application, the Dutch Railways problem. Section 1.5 illustrates the use of cooperating solvers and tightenings to solve the railway problem. Computational results are reported and analyzed.

1.2 DISJUNCTIVE PROGRAMMING

Since our aim is to make continuous and discrete constraint solvers cooperate, we first briefly describe two underlying paradigms of problem representation and problem solving. These are disjunctive linear programming techniques for continuous solvers and constraint satisfaction methods for discrete solvers. Both of these approaches can be understood as instances of the wider area of *disjunctive programming*. Typically, challenging applications addressed by these methods are instances of NP-hard problems. Formulating such problems requires setting up constraints and solving them typically requires nondeterministic search.

1.2.1 DLPs

In $\mathbb{R}^n$, half-spaces and hyperplanes are defined by linear constraints. An intersection of half-spaces in $\mathbb{R}^n$ is called a *polyhedral set.* A *disjunctive set* is a finite union of polyhedral sets, the *disjuncts* of the disjunctive set. The problem of determining whether the intersection of a family of disjunctive sets is nonempty is called a *disjunctive linear program* or DLP.

Suppose P is a DLP given by m disjunctive sets with n disjuncts each. Note that it is possible to assume without loss of generality that all disjunctive sets have the same number of disjuncts; simply add empty disjuncts as required. Moreover, let F_{ij} be the j^{th} disjunct of the i^{th} disjunctive set. Then the goal is to determine whether the set

$$\mathcal{S}(P) = \bigcap_{i<m} \bigcup_{j<n} F_{ij}$$

is nonempty. (The reader will note the analogy with CNF.) The set $\mathcal{S}(P)$ is called the *solution set* of P.

For modeling and computational purposes, one usually starts with an initial feasible region F and works with $F \cap \bigcap \bigcup F'_{ij}$ instead of $\bigcap \bigcup F_{ij}$ where $F_{ij} = F \cap F'_{ij}$. The feasible region F is called the *linear relaxation* of the problem. In the ideal case, F is the convex hull of the solution set and the DLP will reduce to an LP. In a search, the region F is refined and the linear relaxation at each stage represents an approximation to the convex hull of the solution set. If the solution set of a DLP P reduces to the set of all integer points in its linear relaxation, P is called an *integer program* (IP).

The key distinction between a general DLP and an IP is that the DLP requires a mechanism for branching on the disjunctive conditions $\bigcup_{j<n} F_{ij}$ as opposed to one for splitting a problem on a single variable. Disjunctive linear programming, which deals directly with disjunctive requirements without passing to a mixed integer problem with binary variables, has a long history going back to [Glover 65] and [Greenberg 68]. More recently, there has been the work of [Balas 85] and of [Beaumont 90]. Newer developments include [Raman and Grossmann 94] and [Bollapragada et al. 96].

The 2LP language provides a framework for this kind of programming [McAloon and Tretkoff 96]. As we will see below in the Dutch Trains scheduling example, cooperating solvers can be the natural tool for formulating a strategy for solving a DLP.

1.2.2 CSPs

The usual specification of a (finite domain) *constraint satisfaction problem* (CSP) is in terms of a set of *domain variables* $X_1, \ldots, X_m$, each associated with a finite *(initial) domain* D_i of assignable values, and a set of *constraints*, each involving one or more of the domain variables. Constraints can be expressed by relations which limit the possible assignments of values to the domain variables.

A finite domain CSP can also be modeled, in theory, as a DLP by interpreting the domains D_i in the CSP as disjunctive sets whose disjuncts D_{ij} correspond to the finite number of domain values. Conversely, a bounded DLP whose solution set only contains integer points can, again in theory, be defined as a CSP.

A discrete constraint solver uses branching and constraint propagation to solve a CSP. The most commonly used branching technique is *labeling* which assigns domain values to the domain variables. As a simple example of labeling and constraint propagation, we have the Davis-Putnam rules for SAT. One is given a list of propositional clauses and one wants to find a satisfying assignment. A propositional variable can be labeled true or false. When a variable is labeled true (resp. false), all clauses with that variable (resp. its negation) can be deleted from the list of clauses to be satisfied and the negation of the variable (resp. the variable) can be deleted from every remaining clause; when a variable only appears positively (resp. negatively) in the clauses that have not yet been deleted, it can be labeled true (resp. false). If the empty clause is generated by deletions of variables and negated variables, the algorithm backtracks. These propagation rules together with branching determine a sound and complete algorithm for SAT: if all clauses are deleted, the original formula is satisfiable; if satisfiability is not proved, the formula is not satisfiable.

Mathematically sophisticated techniques of constraint propagation and labeling have been developed for SAT and for other CSPs. Recent success stories include the work of Carlier and Pinson on job-shop scheduling problems [Carlier and Pinson 90] and extensions [Caseau and Laburthe 94] and other scheduling applications [Smith and Cheng 93], [Baptiste and Le Pape 97].

1.2.3 Cuts

In both DLP and CSP work, a fundamental technique is the use of redundant constraints which act as catalysts to speed up the solution process. Redundant constraints employed in this way are called *cuts* or *surrogate constraints*. For integer programs, the underlying mathematical theory goes back to Gomory [Gomory 63]; for later developments, see [McAloon and Tretkoff 96] and the references given there.

Let us make the definition somewhat more formally for DLPs. Let P be a disjunctive linear program with n variables and let C be a linear constraint, either an equality ($=$) defining a hyperplane in $\mathbb{R}^n$ or a non-strict inequality ($\leq$ or $\geq$) defining a half-space in $\mathbb{R}^n$. If $\mathcal{S}(P) \cap C = \mathcal{S}(P)$ then C is called a *valid cut* for P. Otherwise, C is called a *non-valid cut*.

If F is the linear relaxation associated with P before performing the cut C, then $F' := F \cap C$ is the new linear relaxation associated with P after performing the cut. If F' is a proper subset of F, then C is called a *strict cut*.

A valid cut is redundant in the sense that it can be added to the disjunctive program without affecting the set of solutions. A non-valid cut, on the other hand, may remove some (or even all) solutions and thus effectively modify the disjunctive programming problem. Non-valid cuts can be useful for *symmetry breaking* to remove "isomorphic" solutions, which only differ from other solutions in that they are permutations of the variable assignments. This is also standard practice in CSPs, e.g. [Van Hentenryck 89]. In optimization problems, solutions can be removed by non-valid cuts as long as some optimal solutions remain, [Hooker et al. 94]. In the rest of this paper we will only consider valid cuts.

Strict cuts shrink the linear relaxation of a disjunctive program and may thus make the problem easier to solve by making the linear relaxation a better approximation to the solution set.

The following example illustrates the usefulness of strict valid cuts in the pigeonhole problem. In this example, as in many others, the valid cuts arise naturally from the semantics of the application.

1.2.4 Example: pigeonhole problem

In the pigeonhole problem m pigeons have to be put into n holes such that only one pigeon is placed in any one hole. Obviously, the problem is solvable iff $m \leq n$, but classical theorem provers as well as discrete constraint solvers based on finite domain constraint propagation may find it hard to show that no solution exists for $m > n$ [Haken 85].

To see this, let p_{ij} be a Boolean variable indicating whether the i^{th} pigeon is in the j^{th} hole. Then

$$\bigwedge_{i<m} \bigvee_{j<n} p_{ij}$$

states that every pigeon has to be put into a hole and the formula

$$\bigwedge_{j<n} \bigwedge_{i<m} (p_{ij} \rightarrow \bigwedge_{k<i} \neg p_{kj})$$

expresses the condition that every hole can contain at most one pigeon. This condition is equivalent to a conjunction of $O(nm^2)$ binary disjunctions $\neg p_{ij} \vee \neg p_{kj}$. Note that for each hole, the disjunctive condition that "at most one of the pigeons can be in hole j" is not expressed as a single formula since propositional logic has no connective for this.

By interpreting the Boolean variables as continuous variables in the interval $[0, 1]$, the pigeonhole problem can be transformed into an integer program whose linear relaxation is defined by the constraints

$$\sum_{j<n} p_{ij} = 1, \qquad i < m$$

$$(1 - p_{ij}) + (1 - p_{kj}) \geq 1, \qquad j < n, i < m, k < i$$

In this form the problem is difficult to solve for integer programming systems, even for small values $m > n$. However in the language of linear constraints on continuous variables, it is possible to express generalized disjunctions naturally. This can be done with the cuts

$$\sum_{i<m} p_{ij} \leq 1, \qquad j < n$$

which reduce the feasible region to the empty set if $m > n$. Therefore the fact that P has no solutions for $m > n$ can be established without performing any search by working on the linear relaxation of the problem alone. For further analysis, see [Barth 96] and [McAloon and Tretkoff 96]; for computational results, see, e.g., [Rodošek et al. 97].

1.2.5 Tightenings

The concept of cuts can be extended with the introduction of additional variables into the problem. A constraint in additional variables which does not change the solution set to a CSP or DLP is called a *generalized cut*. The extension of a CSP or DLP by means of a collection of generalized cuts is called a *tightening*. For DLPs, tightenings are an important tool for making the linear relaxation a closer approximation to the convex hull of the solution set.

Let X be a subset of $\mathrm{I\!R}^{\mathrm{n}}$ and let $k \geq 0$. Then $X \times \mathrm{I\!R}^{\mathrm{k}}$ is called the ($n + k$-dimensional) *cylindrification* of X. The definition of cylindrification is illustrated in Fig. 1.

Let P be a disjunctive program with n variables and with linear relaxation F. Let $\mathcal{C}$ be a conjunction of linear constraints in $\mathrm{I\!R}^{\mathrm{n+k}}$. Let $Cyl(F)$ denote the $n + k$-dimensional cylindrification of F. Then $\mathcal{C}$ is a tightening of P if $\pi(\mathcal{C} \cap Cyl(F)) \cap S(\mathcal{P}) = S(\mathcal{P})$ where π is the projection from $\mathrm{I\!R}^{\mathrm{n+k}} \rightarrow \mathrm{I\!R}^{\mathrm{n}}$ on the first n coordinates.

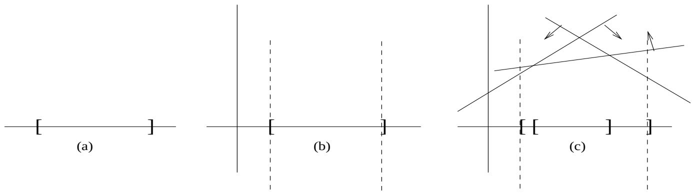

Figure 1.1 A region, a cylindrification and a tightening

The polyhedral set $F' := \pi(\mathcal{C} \cap Cyl(F))$ is the new feasible region associated with P after applying the tightening. If F' is a proper subset of F, then $\mathcal{C}$ is called a *strict tightening*. An example of a strict tightening is shown in Fig. 1c.

Clearly, a collection of cuts for a DLP yields a tightening, simply one which does not introduce any new variables ($k = 0$).

Given a tightening of a DLP which involves additional variables, it is (theoretically) possible to replace it with a set of cuts by computing the projection $\pi(\mathcal{C} \cap Cyl(F))$. Fourier elimination provides an algorithm for carrying this out, e.g. [Schrijver 86], [McAloon and Tretkoff 96]. However, in practice, rather than projecting from $(n+k)$-dimensional space onto $\mathbb{R}^n$, the auxiliary variables introduced by the tightening are kept and a new $(n+k)$-dimensional disjunctive programming problem P' is obtained whose linear relaxation is $\mathcal{C} \cap Cyl(F)$. Any solution to P' yields a solution to P by simply ignoring the additional variables.

The following example illustrates ways of using tightenings to solve a classic crypt-arithmetic puzzle.

1.2.6 Example: crypto-arithmetic puzzle

In the crypto-arithmetic puzzle

`GERALD + DONALD = ROBERT`

there are ten integer variables (`G,E,R,A,L,D,O,N,B,T`). It is straightforward to model this as a CSP. When modeled as a DLP, the initial feasible region can be represented by the polytope F in 10-dimensional space which is defined by

$0 \leq \texttt{G} \leq 9, \ldots, 0 \leq \texttt{T} \leq 9$

$10^5 * \texttt{G} + \ldots + \texttt{D} + 10^5 * \texttt{D} + \ldots + \texttt{D} = 10^5 * \texttt{R} + \ldots + \texttt{T}$

The constraints

$\texttt{D} + \texttt{D} = 10 * \texttt{C} + \texttt{T}, 0 \leq \texttt{C} \leq 1 \qquad (*)$

introduce a new variable, `C`, into the problem, which is to denote the rightmost carry of the addition. The constraints `(*)` yield a tightening of the problem. It is not difficult to compute the feasible region $\pi(\mathcal{C} \cap Cyl(F))$: it is the intersection of the initial feasible region and the half-spaces

`T` $\leq$ `D+D` and `D+D` $\leq$ `T+10` (**)

It is possible to show that these inequalities are strict valid cuts, which implies that the tightening to which they correspond is also strict.

This suggests that one could use the cuts `(**)` instead of the tightening `(*)`. But keeping the additional variable `C` introduced by the tightening is useful because its small domain can help direct the search. Also, there is only one additional multi-linear constraint needed in the 11-dimensional representation whereas two additional multi-linear constraints are needed to describe the new feasible region in the 10-dimensional representation. For more on the DLP case, see [McAloon and Tretkoff 96].

Computational experiments with ILOG Solver for the CSP formulation confirm that the tightening `(*)` is more efficient than the cuts `(**)`. The cuts reduce the number of fails and the computation time by only about 33 %, the tightening reduces the number of fails and the computation time by about 80 % (note that results may differ depending on which heuristic is used).

Tightening a DLP can be critical to solving it. There is a trade-off, however. Tightening, whether with cuts or with constraints on additional variables, increases the number of rows and columns in the matrices that the linear programming engine has to work with. In the example below, we will focus on the trade-off between the matrix size and the number of nodes required to find and verify optimal solutions.

1.3 COOPERATING SOLVERS

In an *integrated approach* to solver cooperation different constraint solvers are used as part of one programming system. This was the case in CHIP. Another integrated approach was developed by DeBacker and Beringer [DeBacker and Beringer 95b] (see also [DeBacker and Beringer 95a]) who obtained promising results using both the discrete and continuous solvers built into ILOG to solve multi-knapsack problems. Depending on the heuristic used for labeling (i.e. assignment of domain values to domain variables), the cooperative approach requires from 25 % to more than 50 % fewer backtracks than the integer programming approach alone (execution times also decrease, but not as impressively), whereas the purely discrete approach is hopelessly slow compared to the two other approaches.

In a *linked approach* the cooperating constraint solvers reside in separate programming systems for which a suitable interface has to be developed. By way of example, a team at Imperial College has linked the discrete constraint logic programming system ECL^{i}PS^e with the commercial MIP solver Cplex. The interface to Cplex is provided as a library to ECL^{i}PS^e [Schimpf and Novello 97]. Results of applying the system to different test cases (including the "progressive party problem" and the pigeonhole problem) are reported in [Rodošek et al. 97].

The advantage of a linked approach is that highly efficient existing systems can be combined. The disadvantage is that the link may not be easy to establish and may be computationally expensive to maintain at runtime. Making libraries of specialized constraint solving tools for both discrete and continuous methods available might be a good way to overcome this technical problem. Our approach using 2LP, which we describe in sections 1.4 and 1.5, has elements of both the integrated and the linked approaches: on the on hand, Cplex is integrated in 2LP as the underlying linear programming solver so that the programmer need not even know that Cplex is used; on the other hand we also establish a link to C which controls the branch-and-bound search on behalf of the discrete solver.

Solnon [Solnon 97] distinguishes between an *active* usage of the linear constraint solver where it is used to reduce the domains of the discrete variables as opposed to a *passive* usage as in DeBacker and Beringer's approach where it was used only to check whether the linear relaxation of an integer programming problem has a solution. The next section shows how even more aggressive use can be made of the linear solver.

1.4 THE DUTCH TRAINS

1.4.1 Background

The Dutch Railways run trains about once an hour from Amsterdam to Vlissingen and from Vlissingen to Amsterdam. These trains are made up of two types of units, one type with 3 cars and one type with 4 cars. The maximum length of a train is 15 cars. Units may be coupled or decoupled at Amsterdam, Rotterdam, Roosendaal, and Vlissingen subject to certain coupling constraints (see below). When a unit is decoupled, it is parked at the station and may be coupled to another train later. At night the cars are stored in one of the four cities mentioned above and maintenance crews work on the cars. For this reason, it is required that the same number of each type of unit be stored in each city every night.

The Dutch Railways want to minimize the number of cars required to satisfy passenger demand; for data see [Schrijver 96]. If only one type of unit is employed and the coupling constraints are not considered, then the problem can be solved quickly as a network flow problem, for which many efficient algorithms are available [Bertsekas 91]. It is interesting to note that, when formulated as a CSP, discrete solvers may find this problem difficult because they do not detect the special network structure that is exploitable. However, when this problem is run as an integer program, the linear relaxation will automatically yield an integer solution because of the underlying geometry of the situation.

When both 3 units and 4 units are used, but coupling constraints are still ignored, we have an integer multicommodity network flow problem which is harder to solve. Here, one cannot simply run an LP in order to get integer solutions as one could in the simpler case with only one type of unit. Using an LP based solver, one must use integer programming and do a search. However, the structure of the problem is such that the optimal solution can be obtained and verified in a very small number of nodes.

But the problem is really more difficult; there are constraints that must be observed when coupling and decoupling units. First, there must be at least one car in a given run from Amsterdam to Vlissingen or from Vlissingen to Amsterdam that makes the entire trip. This is so that passengers need not change cars if they are traveling to the end of the line. In addition, there are restrictions on how and where cars may be coupled or decoupled: At the intermediate stations, Rotterdam and Roosendaal, units may be coupled to the front, and decoupled from the rear of the train. At the terminal stations, Amsterdam and Vlissingen, coupling and decoupling takes place at the same side of the train. This side is the front of the incoming train, which is the same as the rear of the departing train. During each stop of a train either one of the two actions of coupling and decoupling may take place, but not both.

The Dutch Railways have several objectives. Their first and foremost goal, as noted above, is to minimize the number of cars. The next goal is the minimization of the number of units, given the smallest possible number of cars. Finally, given the number of cars and the number of units of each type, the goal is to minimize the total number of kilometers traveled daily by the cars, the "carriage kilometers." (The full problem is thus a "goal program.")

Groot [Groot 96] and Schrijver [Schrijver 96] address the full problem by using polyhedral combinatorics and constraint propagation in a preliminary stage as well as cooperating solvers that do not perform backtracking. They then formulate the problem as an integer program and pass it to the Cplex mixed integer programming library. While quite elegant, there are two issues raised by this approach. First, the number of additional constraints and variables required for the integer programming formulation of the model can surge, depending on the data, to formidable size. Second, the insights into the problem structure coded in the discrete solver are lost when the program is reformulated as an integer program and handed to a general purpose integer programming solver.

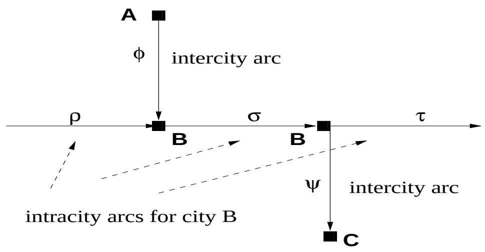

Figure 1.2 Inter- and Intracity Arcs

In this research, we address these issues. To start we expand the discrete solver's functionality and then extend it to work under backtracking in co-operation with the continuous solver. Next we model the problem after the preliminary stage as a disjunctive linear program directly without escalating to an integer programming formulation. This leads to a new kind of algorithm based on the cooperating solvers for the DLP, one that does not require that the problem be formulated as an IP. We then consider tightenings of the DLP and evaluate their impact on the linear relaxation and on node count. The IP formulation emerges as one of the tightenings; at this point, the new algorithm can be compared with various IP algorithms.

1.4.2 Variables and Constraints

Following [Schrijver 96], our model will be built on a network with two types of flow. The two kinds of flow are the movements of the 3 and 4 units, respectively. There are two kinds of arcs in the network: units that go from city to city flow across *intercity arcs* and units that stay at a station waiting to be coupled later to an outgoing train flow across *intracity arcs*.

Fig. 2 illustrates the following scenario: A train leaves city A for city B; this determines the intercity arc ϕ. The arriving arc and the intracity arc ρ from the last train that reached B meet in a **B**-node. The intracity arc σ which connects two **B**-nodes represents the cars waiting in the station between the train's arrival and departure times. The train then departs for city C on intercity arc ψ and train units decoupled in city B are added to those already there to form a new intracity arc τ.

For each arc α in the network, non-negative discrete variables f_α and g_α are introduced where f_α indicates the number of 3 units and g_α the number of 4 units flowing across the arc. For the arcs in Fig. 2 we have the following flow constraints:

$f_\phi + f_\rho = f_\sigma \qquad g_\phi + g_\rho = g_\sigma$

$f_\sigma = f_\tau + f_\psi \qquad\qquad g_\sigma = g_\tau + g_\psi$

The intercity arcs will be called *legs*. For each leg λ the initial domain of f_λ is 0..5 and that of g_λ is 0..3 because of the 15 car restriction. For the intracity arcs, the initial domains are more extensive but are quickly cut down to size as things proceed.

For each leg λ we have a discrete variable $comp_\lambda$ whose initial domain is the set of all possible arrangements of units. Since there can be at most 15 cars on any train there are 30 possible values $3, 4, 33, \ldots, 33333, 4443$ in order. If c is a composition, we denote by $x(c)$ the number of 3 units in c and by $y(c)$ the number of 4 units.

We thus have the functional dependencies $f_\lambda = x(comp_\lambda)$ and $g_\lambda = y(comp_\lambda)$ which correspond to a collection of constraints such as

$comp_\lambda = 334 \rightarrow f_\lambda = 2$

$comp_\lambda = 334 \rightarrow g_\lambda = 1$

$f_\lambda = 3 \rightarrow comp_\lambda \in \{333, 3334, 3343, 3433, 4333\}$

$g_\lambda = 3 \rightarrow comp_\lambda \in \{444, 3444, 4344, 4434, 4443\}$

For each pair of successive legs λ_1 and λ_2 the rules for the coupling and decoupling of trains are enforced by coupling constraints, e.g. going from Rotterdam to Roosendaal (λ_1) and then from Roosendaal to Vlissingen (λ_2) we have

$comp_{\lambda_1} = 343 \rightarrow comp_{\lambda_2} \in \{343, 34, 3, 3343, 4343\}$

These constraints give rise to a propagation mechanism. The set of intercity arcs is partitioned by the schedule into seven sequences with the property that there is one intracity arc connecting every pair of adjacent arcs in the sequence. For each sequence and each leg, a value c can be eliminated from the domain of $comp_\lambda$ if there is no sequence of arcs starting at the first leg of the sequence and reaching the final leg which passes through c.

For the intercity arcs we also have data on first and second class demand which yield capacity coefficients $cap_{first,\lambda}$ and $cap_{second,\lambda}$ and demand constraints:

$3 \cdot cap_{first,\lambda} \cdot f_\lambda + 4 \cdot cap_{first,\lambda} \cdot g_\lambda \geq first_\lambda$

$3 \cdot cap_{second,\lambda} \cdot f_\lambda + 4 \cdot cap_{second,\lambda} \cdot g_\lambda \geq second_\lambda$

For each trio λ_1, λ_2, λ_3 of legs that correspond to the stops on a trip from one terminus to the other we have a continuity constraint:

$f_{\lambda_1} + g_{\lambda_1} - f_{\lambda_2} - g_{\lambda_2} + f_{\lambda_3} + g_{\lambda_3} \geq 1$

which, given the coupling constraints, guarantees that at least one car will make the entire journey. These constraints can be used in conjunction with bounds on the intracity arcs for further constraint propagation and then sharpened.

Finally we must formulate the objective functions for the three goals of the problem. Note that the total number of units of each type required is given by the flow across the intracity arcs that correspond to the overnight wait.

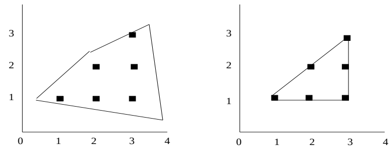

Figure 1.3 The convex hull of integer points

Denoting the overnight intracity arcs by the names of the cities, am, rm, rl and vn, we obtain a total number of cars z_1:

$$z_1 = 4 \cdot (g_{am} + g_{rm} + g_{rl} + g_{vn}) + 3 \cdot (f_{am} + f_{rm} + f_{rl} + f_{vn})$$

Similarly, the number of units z_2 required is

$$z_2 = g_{am} + g_{rm} + g_{rl} + g_{vn} + f_{am} + f_{rm} + f_{rl} + f_{vn}$$

The total number of carriage kilometers z_3 is

$$z_3 = \sum_{\lambda} km_{\lambda} \cdot (3 \cdot f_{\lambda} + 4 \cdot g_{\lambda})$$

where km_{λ} is the length of λ in kilometers. All this defines a CSP, albeit one very difficult to solve in this form. Let us call this problem P_1. At this point continuous variables are introduced. Each f_{α} has a doppelgänger F_{α}, and each g_{α} its G_{α}. (In spirit this is close to the way cooperating solvers are set up in ILOG Planner [ILOG 96].) Demand, flow, continuity and bounding constraints can be placed on these variables. These are all linear constraints. Similarly, each z_i has its Z_i and the objective functions are defined by linear constraints.

The demand constraints can be sharpened considerably by means of a device from polyhedral combinatorics. Fix a leg λ and consider the demand and capacity constraints on F_{λ} and G_{λ} and the region they define in 2-dimensional space. From the domains of f_{λ}, g_{λ} and $comp_{\lambda}$, we have a set of integer points in this region; the original region defined by the demand constraints can then be replaced by the convex hull of these integer points as in Fig. 3.

Next constraint propagation on the domains of f_{λ} and g_{λ} can be done by testing each possible candidate pair of such values for F_{λ} and G_{λ} for feasibility in the linear relaxation. For all legs λ, the domains of f_{λ}, g_{λ} and $comp_{\lambda}$ can be shrunk this way; the original two dimensional hulls can now be replaced with tighter hulls and the process can be re-iterated until a fixed point is reached. In practice, the CWI team did their version of this at most twice; for the benchmarks below, we continue until a fixed point is reached. During this iterative process, upper and lower bounds on the intracity arcs are tightened also; this is an example of what is called *active use* of the LP solver by [Solnon 97].

A remark on the way things work in practice: Values for z_1 and z_2 are determined without enforcing the coupling constraints. As noted above, this can be done rather easily using integer programming techniques. With these values in hand, the third goal and the coupling constraints are addressed and the techniques of cooperating solvers and polyhedral combinatorics applied. If no solution can be found, the code must backtrack to revise z_2 and possibly z_1 and try again. With the given data, this deep backtracking is never required. The reason for approaching the application this way is that fixing the values of z_1 and z_2 has a substantial impact on the effectiveness of the polyhedral combinatorics and the cooperating solvers.

1.5 ALGORITHMS AND TIGHTENINGS

1.5.1 An Algorithm

The problem now is to find values for the f_α, F_α and g_α, G_α such that the constraints on the discrete variables and on the continuous variables are feasible. From the point of view of the continuous variables, this is a disjunctive programming problem. By that we mean, in addition to the integrality requirements, the linking and composition constraints on intercity variables f_λ, g_λ and $comp_\lambda$ determine further conditions on the continuous variables F_λ, G_λ all of which form a DLP. Let us call this problem P_2. Any solution to the CSP P_1 yields a solution to the DLP P_2 and conversely any solution to P_2 yields a solution to P_1. We are now going to develop a cooperative algorithm for solving this symbiotic pair of problems.

The algorithm will be a branch-and-bound search, to find better and better solutions until the optimal one is found and its optimality verified. Placing oneself *in medias res*, analysis of a node in the search begins with optimization of the objective function in the linear relaxation. This yields a solution to the relaxed problem called the *witness point*.

If the witness point at a node is an integral point, it provides candidate values for all the variables f_α, g_α which will, of course, satisfy the flow, demand and continuity constraints. If, for every leg λ, there is a composition c in the current domain of $comp_\lambda$ such that $f_\lambda = x(c)$ and $g_\lambda = y(c)$, then a solution has been found. We say that the witness point has been *lifted* to a solution of the full problem. Moreover this node can be fathomed since further development of the node can not lead to a better solution.

If the witness point has non-integral coordinates, a variable F_α or G_α is chosen from these as the branching variable. The one chosen will be one which has maximal effect on the objective function as determined by a heuristic which takes into account the length of the arcs measured in kilometers.

When a variable is branched upon, the technique of [Land and Doig 60] is used in coordination with domain information from the discrete solver. This way convexity is used to detect the infeasibility of a range of values by going through the domain values in increasing (resp. decreasing) numerical order, e.g. [McAloon and Tretkoff 96]. With the Land and Doig technique, the variable branched upon will be fixed at all nodes below the current node. When a continuous variable is fixed this way in the branching process, this labels its discrete doppelgänger; the propagation from this labeling is then used to restrict the domains of both the discrete and continuous variables. This is an example of the more active usage of the cooperating solvers which we mentioned at the end of section 1.3. Moreover, if the discrete constraint propagation detects infeasibility, backtracking of both the continuous and discrete engines is triggered. Similarly, when the continuous solver detects infeasibility, backtracking takes place on both sides.

Now suppose that the witness point has all integral coordinates but that there is no way to lift it to a full solution by finding values for the variables $comp_\lambda$ consistent with the values the witness point yields for f_λ and g_λ. At this point the discrete solver is invoked to find a pair of adjacent legs λ', λ'' so that all compositions obtained by lifting the witness point values of $f_{\lambda'}$, $g_{\lambda'}$ and $f_{\lambda''}$, $g_{\lambda''}$ cannot be made consistent. A loop invariant argument shows that at least one of the variables $f_{\lambda'}$, $g_{\lambda'}$, $f_{\lambda''}$, $g_{\lambda''}$ is not fixed yet (i.e. its domain contains at least two values) and this variable can be chosen for branching.

This algorithm is quite different from the branch-and-bound algorithms of integer programming, most obviously, in that it requires splitting the problem into subproblems by branching even if the witness point is integral. Thus, verification that the algorithm is sound and complete also requires an analysis of the discrete component. In sum, the use of cooperating solvers is critical for coding a solution to this DLP.

The characteristic of this DLP formulation of the problem and its accompanying algorithm is that the minimal amount of linear programming machinery is required. This makes it very good at diving for solutions but its looseness weakens its ability to verify optimality. Its performance is listed along with that for the next two formulations in Tables 1a, 1b and 1c. (For the benchmarks presented in the tables the data of the Dutch Railways correspond to the rows in the tables labeled 100 %. The other data sets are obtained by multiplying the original passenger load data by the given percents.) In fact, one can see from Table 1b that on certain data the solution requires a relatively large number of nodes. We next address this by applying tightenings to the model.

1.5.2 Valid Tightenings

We now describe two basic valid tightenings of the disjunctive programming problem P_2. The algorithm described in the previous section will apply without change to these tightened versions. Finally we will discuss alternative algorithms for the second tightening, which in effect is an IP formulation of the problem.

The first tightening is obtained by adding variables for each pair of values for f_λ and g_λ for intercity arcs λ. Thus if c is a composition in the domain of $comp_\lambda$ and $u = x(c)$ and $v = y(c)$, we add a continuous variable $W_{\lambda,u,v}$, with bounds 0 and 1. This variable will serve as a "fuzzy boolean"; if it is 1, we will have $f_\lambda = u$ and $g_\lambda = v$. Three kinds of constraints are added. First we express the generalized disjunction to the effect that exactly one of the pairs u,v must be chosen for each leg. Then we express the flows across the leg as linear combinations of the $W_{\lambda,u,v}$. Finally, we express the fact that the values of the $W_{\lambda,u,v}$ link from leg to leg.

- $\sum_{(u,v)} W_{\lambda,u,v} = 1$
- $F_\lambda = \sum u \cdot W_{\lambda,u,v} \quad G_\lambda = \sum v \cdot W_{\lambda,u,v}$
- For adjacent legs λ, λ',
  $W_{\lambda,u,v} \leq \sum_{(u',v')} W_{\lambda',u',v'} \quad \sum_{(u,v)} W_{\lambda,u,v} \leq W_{\lambda',u',v'}$,
  where u, v is consistent with u', v'

From the logic of the problem, it is clear that these constraints define a valid tightening. To see that the tightening is strict, note in Table 1b that the minimum of the linear relaxation goes up from the loose to the tighter model (except for 100 %). It is possible for the witness point to be integral without providing a solution to the DLP. It can also happen that the witness point is not integral in all coordinates but still provides a solution to P_2. Again, coding a solution to this DLP without cooperating solvers would be very difficult. As can be seen in Table 1b, there is a trade-off between matrix size and node count between this and the previous model. Table 1c shows that computation times sometimes decrease (by up to 70 %), but there is also the anomaly of the case of 105 % where the time doubles. For 101–104 % the tighter model leads to a significant decrease in the time needed for the final search phase.

The second tightening is obtained by adding a continuous variable $V_{\lambda,c}$ for every value in the domain of $comp_\lambda$ with $0 \leq V_{\lambda,c} \leq 1$. Again three kinds of constraints are added:

- $\sum_c V_{\lambda,c} = 1$

- $F_\lambda = \sum x(c) \cdot V_{\lambda,c} \qquad G_\lambda = \sum y(c) \cdot V_{\lambda,c}$

- For adjacent legs λ, λ',

  $V_{\lambda,c} \leq \sum_{c'} V_{\lambda',c'} \quad V_{\lambda',c'} \leq \sum_c V_{\lambda,c}$
  where c and c' satisfy the coupling constraints

We have thus rewritten the logic of the coupling constraints in terms of the variables $V_{\lambda,c}$. This gives a valid tightening. Comparing the "lmin" columns in Tables 1a and 1b, we can see that this is a strict tightening and one that is even tighter than before. We can also see from Table 1a that the above algorithm performs even better in terms of node count than it did with the previous tightening, as is to be expected. It is interesting to note once again that applying our algorithm to the model in this form illustrates that it is possible for the witness point to yield a solution to P_2 without all the variables $V_{\lambda,c}$ being integral. Again we save time in those cases (esp. 101–105 %) where the times for the final search phase in the loose and tighter models were non-trivial.

The timings for all 3 models up to 100 % and for the tightest model up to 105 % are better than those reported in [Groot 96], although admittedly such comparisons are difficult to make across platforms.

Table 1a. Benchmarks

			tightest					
%	3s/4s	kms	rows	cols	lmin	nd	SOS	Cpx
95	11/4	46998	1468	823	46998	1	74	348
96	11/4	47148	1359	745	47148	1	74	348
97	14/2	48716	1277	693	48444	3	74	348
98	14/2	48902	1382	758	48617	4	94	108
99	14/2	48902	1382	758	48617	4	94	108
100	14/2	49766	962	473	49766	1	1	4
101	13/3	48880	1820	857	48723	2	8	76
102	12/4	49058	2352	1342	48799	36	231	20
103	12/4	49574	2199	1263	49441	7	92	91
104	11/5	49652	2525	1454	49598	15	92	91
105	11/5	50780	2219	1281	50603	8	92	91

Table key: Passenger demand percentage, numbers of 3 and 4 units, kilometers in optimal solution, number of rows, number of columns, km minimum for linear relaxation, nodes (all for tightest model), nodes for SOS algorithm, nodes given by Cplex IP.

Table 1b. Benchmarks continued

	tighter				loose			
%	rows	cols	lmin	nd	rows	cols	lmin	nd
95	1163	656	46998	1	544	382	46848	94
96	1107	611	47148	1	534	359	46998	11
97	1147	622	48090	12	544	380	48004	13
98	1199	655	48434	2	545	380	48352	6
99	1199	655	48438	2	545	380	48352	6
100	907	442	49766	1	501	275	49766	1
101	1458	839	48588	164	595	471	48258	532
102	1706	974	48460	213	636	497	47527	1167
103	1601	922	49430	16	617	487	48654	1308
104	1740	1002	49281	76	631	500	48582	1502
105	1585	912	49841	4962	610	477	49210	8661

Table key: Passenger demand percentage, number of rows, number of columns, km minimum for linear relaxation, nodes (first for tighter model, then for loose model)

Table 1c. Computation times

	loose		tighter		tightest	
%	total	search	total	search	total	search
95	105.1	16.0	88.7	0.5	90.2	1.0
96	69.5	1.5	65.9	0.4	68.2	0.8
97	46.7	3.3	51.8	5.9	46.8	1.7
98	44.1	1.2	46.2	1.1	52.7	6.4
99	43.4	1.0	45.2	1.1	51.5	6.9
100	34.9	0.2	33.0	0.3	34.2	0.3
101	208.1	127.5	115.7	49.2	74.1	6.5
102	441.3	276.8	199.1	96.3	174.3	72.8
103	389.3	276.7	102.4	14.6	107.2	21.4
104	653.0	505.9	258.0	127.0	222.1	95.5
105	1693.1	1573.6	3321.7	3216.3	139.1	33.8

Table key: Passenger demand percentage, total times and times for final search phase for loose, tighter and tightest models (all times in seconds on a Sun UltraSparc).

However, when $V_{\lambda,c}$ is integral, if it has value 1, then $comp_\lambda = c$; if it has value 0, then $comp_\lambda \neq c$. Moreover, since values for the variables $comp_\lambda$ uniquely determine values for the f_λ and g_λ, a witness point at which all the variables, including the $V_{\lambda,c}$ are integral provides a solution to the full DLP. In short, we have formulated the problem as an IP. Therefore, with this new version, the model can also be handed over to an IP solver such as the one in the Cplex library; the node counts generated by Cplex are also given in Table 1a.

Moreover, with this form of the model, we can take advantage of the special structure of the constraints $\sum_c V_{\lambda,c} = 1$. These "sum to 1" constraints are known as *specially ordered sets* (SOS) and a branching strategy can be formulated which treats the summands in batches. In fact, this is another example of cooperating solvers, but one which is already integrated into most IP systems; for a description of the mechanics involved, see [McAloon and Tretkoff 96]. Node counts obtained by this approach are given in Table 1a.

Another classical example of cooperating solvers built in to an IP solver is the handling of "semi-continuous" variables, e.g. Xpress-MP. Yet another is the pre-solving which is done in most IP codes, e.g. [Savelsbergh 94]. What we see here is that for a coherent view of both the IP approach and the DLP approach, cooperating solvers emerge naturally.

It should be noted that if pre-solving is turned off, the performance of the Cplex IP solver on these benchmarks degrades considerably. On the other hand, pre-solving has not yet been implemented in our code where the new DLP algorithm (or the SOS algorithm) is applied. Clearly, this is an area that should be explored.

1.6 CONCLUSIONS

Our analysis was motivated by questions raised by Schrijver as to whether constraint programming could provide an assist in dealing with the Dutch Railways problem. We found that, indeed, cooperating solvers led to new formulations of the problem and a new species of algorithm to solve it. There are numerous other families of applications which might best be served by cooperating constraint solvers. Examples include absolute deviation based portfolio selection models and parallel machine scheduling.

For the particular application of this paper and algorithms to solve it, several lines for improvement suggest themselves; for example, a multicommodity flow relaxation of the problem could be used both for heuristics and for pruning the search space. Another is to use the original DLP version in conjunction with randomized "shuffle" algorithms. Yet another is to experiment with alternatives to depth-first search based on iterative deepening.

Acknowledgements

The authors would like to thank Krzystof Apt and Lex Schrijver for introducing us to the problem of the Dutch Railways and for stimulating discussions on constraint programming. We would also like to thank CWI for their scientific hospitality. This research is supported in part by ONR grant N00014-96-1-1057.

References

[Balas 85] Balas, E.: Disjunctive Programming and a Hierarchy of Relaxations for Discrete Optimization Problems. *SIAM Journal Algebraic and Discrete Methods* **6** (1985), 149–156

[Baptiste and Le Pape 97] Baptiste, P.; Le Pape, C.: Constraint Propagation and Decomposition Techniques for Highly Disjunctive and Highly Cumulative Project Scheduling Problems. Submitted to CP97.

[Barth 96] Barth, P.: *Logic Based 0-1 Constraint Programming.* Kluwer 1996

[Bertsekas 91] Bertsekas, D.: *Linear Network Optimization: Algorithms and Codes.* MIT Press 1991

[Beaumont 90] Beaumont, N.: An algorithm for disjunctive programming. *European Journal of Operational Research* **48** (1990), 362–371

[Bollapragada et al. 96] Bollapragada, S.; Ghattas, O.; Hooker, J. N.: Optimal Design of Truss Structures by Logic-Based Branch and Cut, to appear

[Carlier and Pinson 90] Carlier, J.; Pinson, E.: An algorithm for solving the job shop problem. *Annals of Operations Research* **26** (1990), 269–287

[Caseau and Laburthe 94] Caseau, Y.; Laburthe, F.: Improved CLP Scheduling with Task Intervals. In: Van Hentenryck, P. (ed.): *Proc. of the 11^{th} Int. Conf. on Logic Programming*, 369–383, MIT Press 1994

[Colmerauer 87] Colmerauer, A.: Opening the Prolog III Universe. *BYTE Magazine*, 8/1987

[DeBacker and Beringer 95a] DeBacker, B.; Beringer, H.: Combinatorial Problem Solving in Constraint Logic Programming with Cooperating Solvers. In: Beierle, C.; Plümer, L (eds.): *Logic Programming: Formal Methods and Practical Applications*, Elsevier 1995

[DeBacker and Beringer 95b] DeBacker, B.; Beringer, H.: Cooperative solvers and global constraints: the case of linear arithmetic constraints, *ILPS'95 Workshop*, 1995

[Dincbas et al. 88] Dincbas, M.; van Hentenryck, P.; Simonis, H.; Aggoun, A.; Graf, T.; Berthier, F.: The Constraint Logic Programming Language CHIP. *FGCS'88 Proceedings* (1988) 693–702

[Glover 65] Glover, F.: A Bound Escalation Method for the Solution of Integer Linear Programs. Carnegie-Mellon University, 1965

[Gomory 63] Gomory, R. E.: An Algorithm for Integer Solutions to Linear Programs. In: Graves, R. L.; Wolfe, P.: *Recent Advances in Mathematical Programming*. McGraw-Hill (1963) 269–302

[Greenberg 68] Greenberg, H.: A Branch-and-Bound Solution to the General Scheduling Problem. *Operations Research* **8** (1968), 353–361

[Groot 96] Groot, R.: Minimum Circulation of Railway Stock — An Integer Programming Problem. MS Thesis, University of Amsterdam, 1996

[Haken 85] Haken, A.: The Intractability of Resolution. *Theoretical Computer Science* **39** (1985) 297–308

[Hooker et al. 94] Hooker, J. N.; Yan, H.; Grossmann, I. E.; Raman, R.: Logic Cuts for Processing Networks with Fixed Charges. *Computers Ops. Res.* **21** (1994) 265–279

[ILOG 96] ILOG Planner Reference Manual, ILOG 1996.

[Jaffar and Michaylov 87] Jaffar, J.; Michaylov, S.: Methodology and Implementation of a CLP System. In: Lassez, J.-L. (ed.): *ICLP'87 Proceedings*. MIT Press 1987

[Land and Doig 60] Land, A. H.; Doig, A. G.: An Automatic Method of Solving Discrete Programming Problems. *Econometrica* **28** (1960) 497–520

[McAloon and Tretkoff 96] McAloon, K.; Tretkoff, C.: *Optimization and Computational Logic*. John Wiley & Sons 1996

[McAloon and Tretkoff 97] McAloon, K.; Tretkoff, C.: Logic, Modeling, and Programming. *Annals of Operations Research*, to appear

[Raman and Grossmann 94] Raman, R.; Grossmann, I. E.: Modeling and Computational Techniques for Logic-Based Integer Programming. *Computers and Chemical Engineering* **18** (1994), 563–578

[Rodošek et al. 97] Rodošek, R.; Wallace, M.; Hajian, M. T.: A New Approach to Integrate Mixed Integer Programming with CLP, Technical Report, Imperial College

[Savelsbergh 94] Savelsbergh, M.: Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing* **6**, 445–454

[Schimpf and Novello 97] Schimpf, J.; Novello, S.: IC-Parc ECL^{i}PS^e Library Manual, Imperial College

[Smith and Cheng 93] Smith, S. F.; Cheng, C.-C.: Slack-Based Heuristics for Constraint Satisfaction Scheduling. In: *Proceedings of the Eleventh National Conference on Artificial Intelligence*, 1993

[Solnon 97] Solnon, C.: Coopération de solveurs linéaires sur les réels pour la résolution de problèmes linéaires sur les entiers. To appear in *Journées Francophones 97*

[Schrijver 86] Schrijver, A.: *Theory of Linear and Integer Programming*, John Wiley & Sons 1986

[Schrijver 96] Schrijver, A.: Minimum Circulation of Railway Stock, *CWI Quarterly* 1996

[Van Hentenryck 89] Van Hentenryck, P.: *Constraint Satisfaction in Logic Programming.* MIT Press 1989

[Wetzel 97] Wetzel, G.: Abductive and Constraint Logic Programming. PhD Thesis, Imperial College