2LP: Linear Programming and Logic Programming

Ken McAloon and Carol Tretkoff
Logic Based Systems Lab
CUNY Graduate Center and Brooklyn College
2900 Bedford Avenue
Brooklyn, NY 11210
mcaloon@sci.brooklyn.cuny.edu
tretkoff@sci.brooklyn.cuny.edu

The 2LP system is a step in the "Operatica Program," a project whose grand design is to provide an elegant and powerful programming language environment for combining AI and OR methods for decision support software systems. The term "Operatica" was coined by J.L. Lassez to suggest an analogy with the Mathematica system which provides a programming environment for symbolic mathematical computation. In the dialogue between AI and OR, there are two basic themes: (1) declarative programming and the notion of logical consequence and, (2) procedural programming and the search algorithm in its many variations. Integrating AI and OR requires an environment that combines a modeling language with a logic based language. 2LP, which stands for "linear programming and logic programming," has the simplex based search mechanism of linear programming and the backtracking mechanism of logic programming built in. 2LP is both an algebraic modeling language and a logical control language. By bringing these techniques together in a language which has standard C style syntax and treats the mathematical module in an object-oriented way, this technology provides very powerful and usable tools for decision support programming.

The design decisions that led to 2LP were based on the following considerations:

- For run time efficiency the system should enforce a restriction to linear constraints at compile time.
- The array rather than the linked-list is the natural data structure for mathematical modeling.
- Communication between the logic and the numerical solver should be primitive to the language.
- A small language with standard syntax and with explicit integration of procedural programming constructs would be accessible to decision support programmers.
- The power of constraint programming is such that a compact language is sufficient for the intended applications.
- The system should be callable as a set of library routines.
- Hooks for expandability to an or-parallel system should be built in to the system.

In this paper, after discussing the relationship of 2LP to other programming paradigms and introducing some of the features of the language, we present some examples of 2LP applications and discuss future directions. Other applications can be found in [McAloon,Tretkoff 2], [McAloon,Tretkoff 3].

1. 2LP and other programming paradigms

2LP introduces a new built-in data type **`continuous`** in order to capture the modeling methods of linear programming. Constraints on these variables define a polyhedral set. The representation of this convex body includes a distinguished vertex, called the *witness point*. Other structural information and communication routines are also connected with the polyhedral set. This collection forms an "object" to use object-oriented programming (OOP) terminology. 2LP is a "little language" that supports these constructs and has its own interpreter. On the other hand, object-oriented programming as in C++ enables one to extend a language with new data types without writing an interpreter for the expanded language. Because of the need for logic based control, backtracking and restoring of state in 2LP, the C++/OOP method can not be used for the purposes for which 2LP is designed; a little language approach proves necessary.

There has long been a debate in the AI world over the relative importance of declarative versus procedural programming. 2LP enables the programmer to make clean distinctions between the declarative and procedural aspects of a model and facilitates combining them for effective problem solving. The CLP languages Charme and CHIP integrate a declarative treatment of constraints with integrality and logical requirements in a larger programming framework and are an important development. On other hand, in CLP(**R**), in Sicstus Prolog with constraints etc., the programmer must provide the routines for driving continuous variables to integer values or to ranges that satisfy given logical requirements. The same is true of 2LP. In 2LP, this uncommitted approach is turned into a feature and tools are provided so that the programmer can tailor the problem solving method to the application at hand. These tools enable the modeler to use fuzzy logic and insight provided by the geometry of the polyhedral object defined by the constraints. This also means that the modeler can utilize powerful OR techniques that are used in MIP solvers such as Specially Ordered Sets of various types. In a similar vein, Prolog III provides built-in facilities which control search for an integer solution to constraints via communication with the state of the solver.

2LPis different from algebraic modeling languages such as AMPL [Fourer,Gay,Kernighan] in two important ways. First it is a logical control language as well as a modeling language. Second, 2LP supports parameter passing and procedure based modularity. In this it differs from current modeling languages in which the program basically consists of global variables and loops to generate constraints. On the other hand, AMPL can support network models directly and supports a set-theoretic view of a model as well as an algebraic view. AMPL also supports symbolic representation of indices and provides powerful and original logical/arithmetic connectives and quantifiers that operate at the level of the description of a constraint. Another feature of AMPL is the way it can enforce integrity constraints on the parameters of the model; the linkage between the data (concrete and large) and the model (abstract and small) is a priority in the AMPL system. Both 2LP and AMPL have a “symbolic computation” facility that recognizes that linear combinations of linear expressions are linear; this means that the modeler can employ natural mathematical expressions in constraints and is not forced to express constraints in some “canonical” form.

The 2LP language has evolved out a series of smaller languages which were developed step by step to maintain the best balance among the logic module of the language, the linear programming module and the intended applications. The decision to restrict the mathematical solver to linear constraints was based in part on the fact that the atomic operations of a programming language should be efficient. This was also motivated by the fact that the range of applications that can be handled with the combination of logic and linear constraints is remarkably rich. The theoretical basis for this work was developed in [Cox,McAloon,Tretkoff] which contains an analysis of the CLP scheme in terms of the computational complexity of the “halting problem” of languages which are instances of the scheme. The analysis also calibrates the role of the underlying logic used - propositional, relational, functional. Let D denote the ordered additive group of the real numbers as a **Z**-module. Then in the terminology of that paper, implementations of minimal-CLP(D) and conservative-CLP(D) were done before the current version of the language was designed and implemented. The minimal and conservative languages are based on propositional and relational logic respectively. The logic of full 2LP has a more complex character. All along an important consideration was the challenge of compiling this new kind of programming language building on the work of [Warren], [Jaffar,Michaylov] and others.

As a language for AI applications 2LP is dramatically different from LISP and Prolog. For classical AI applications, LISP and Prolog have certain definite strengths. Both support symbolic data and very flexible handling of types. Each is built on one mighty algorithm - lambda conversion in the case of LISP and unification in the case of Prolog - which brings a uniformity and declarative quality to programming [Lassez]. Both make programming with recursion very natural and recursion is the control mechanism of choice when working with data and data structures whose semantics are free in the mathematical sense. This is in contrast to classical imperative languages which are oriented toward numerical processing with its well-understood underlying mathematical semantics. Moreover, the remarkable transparent memory management of LISP and Prolog, made possible by the role of the linked-list as the basic data structure, makes writing code for various search strategies very elegant in these languages: the stacks and queues are managed by the system and garbage is collected in the background. To this Prolog adds its built-in logical control and more declarative programming facility. The weak point, if you will, of these languages historically has been their handling of numerical computation. The development of the CLP languages was motivated to a large extent by the aim of integrating numerical computation naturally into the declarative framework of logic programming. These systems have been very successful.

Traditionally, the language of choice for heavily numerical computing is FORTRAN. Though built on a static picture of memory management based on the array, FORTRAN has been in fact used for many interesting systems which link AI and mathematics. The language C occupies a middle position and supports arrays and pointers which allow for an elegant treatment of linked-lists which captures much of the modeling power of LISP and Prolog. 2LP itself does not support pointers, and if queues, stacks or linked-lists are used in a 2LP program, they are coded in the FORTRAN style. However, such data management, especially if it is a computationally significant part of the application, is best done in C itself and handled by means of calls from 2LP to external C functions. The fact that 2LP follows the C picture of memory allocation, parameter passing and overall arithmetic syntax makes this integration of C functions very natural.

2. The 2LP language

The basic data types of 2LP are **`int`**, **`double`** and **`continuous`**, which is a new type for working with constraints. The data structures are arrays of elements of a given basic type. Arrays are the natural data structure for mathematical programming and facilitate working with constraints. In a 2LP program, variables of type **`continuous`** function in the declarative manner of the variables of linear algebra or linear programming. As a program progresses, linear constraints on the **`continuous`** variables are generated and define an evolving polyhedron in n-dimensional space, where n is the number of **`continuous`** variables declared in the program.

While the fundamental operation on variables of type **`int`** and **`double`** is assignment the **`continuous`** variables in a 2LP program are constrained by linear equalities and inequalities. Thus 2LP extends to this new type the arithmetic operators +,- and * (for coefficients) and the relational operators **`==`**,**`<=`** and **`>=`**. The = operator is used for assignment as in C. The operators **`!=`**,**`<`** and **`>`** are also extended to continuous variables and are interpreted by means of negation-as-failure. By default **`continuous`** variables are constrained to be non-negative - unless a negative lower bound is explicitly imposed. A program consists of declarations of global storage including external storage and external functions, a main procedure called **`2lp_main()`** and other procedures. All storage locations created in the program are either global variables, external global variables or are declared in **`2lp_main()`**; the exception is that loop control variables are introduced locally. Procedures take parameters by value or by reference following C and C++ conventions. Loop variables can only be passed by value; **`continuous`** variables can only be passed by reference. Variables which are passed by value to a procedure can not be assigned in that procedure. As in logic programming, procedures may contain disjunctive control constructs which generate “choice points” in the computation. These are nodes in the logic programming search to which the program can return later in its run. The system uses depth-first search and chronological backtracking. Upon “failure” the backtracking mechanism resets the constraints and the continuous variables to their state at the previous choice point; loop variables and variables passed by value are also reset to their values at the choice point. The state of the global variables of type **`int`** and **`double`** and of the variables of these types passed by reference is left unchanged by backtracking. In classical logic programming, the “logical variable” is not assigned values but is monotonically constrained by unification. Thus the **`continuous`** variables, the loop variables and the variables passed by value capture the role of the “logical variable” and provide the system its declarative kernel.

As a programming language, 2LP is small; its ambitions are circumscribed and focused. It is designed to be embedded in the larger programming scheme of things and to be used as a set of library routines. Arrays of **`double`**s and **`int`**s and C-functions can be provided by the principal system and addressed as **`extern`** by the 2LP model. The language is built on an abstract machine model called the S-CLAM which is an analog of the WAM and CLAM in the smaller 2LP context [McAloon,Tretkoff 1]. For the record, the S-CLAM supports tail recursion and classical logic programming formats such as multiple procedure definitions and cut. The 2LP architecture provides for a very modular linkage between the logical control and the mathematical constraint solver. As a result, 2LP supports use of optimization libraries such as Cplex and OSL as well as its own simplex based mathematical code. This means that 2LP can be used for applications that require state-of-the-art optimization software and the system can move with changes in the mathematical programming world. (It is also to be hoped that by interacting with systems like 2LP, optimization software will become more responsive than at present to requirements like incremental loading and deleting of constraints.)

2LP provides language tools for programming the desired semantics for an optimization application. For a classical linear program the semantics of optimization are straightforward. When a richer mix of programming constructs and logic is introduced, there are many possible interpretations to "max: ... subject_to ... ." In particular, the programmer will want to control the state of the geometric object defined by the constraints both during and upon exiting this block of code. For that reason, 2LP supports control mechanisms and some built in optimization constructs for writing code that expresses the semantics intended by the modeler. This is especially helpful when optimization routines occur as subroutines in a larger application.

The logic programming "negation-as-failure" is supported by 2LP. It turns out that in programming with constraints, it is double-negation-as-failure that is a most powerful and useful tool. The reason for this is simple: a call to **`!!add_constraints()`** is a test for the consistency of the constraints generated by a call to **`add_constraints()`** with the current geometric object; this test for consistency leaves the object unchanged. It is this type of query that is needed for Land and Doig search, for A* search, for computing heuristics, for lookaheads etc. Again for the record, 2LP interprets the open comparison operators **`<`**,**`>`**, and **`!=`** by means of negation-as-failure. This is a coherent reading which preserves the closed, convex set interpretation of constraints and which gives these open operators a natural role to be used as guards or checks.

2LP can be used as a stand-alone system or as a callable library. In stand-alone mode, a 2LP model is sent directly to the compiler/interpreter. In library mode, 2LP is called from C as a function; the 2LP function is sent the whereabouts of the data and C functions it is to use as well as the 2LP model; the model can be sent either as a file or as a pre-compiled C data structure.

3. Dynamic alternation

The 2LP system takes a structured approach to logical modeling. It supports **`and`**-loops, **`or`**-loops and other logical constructs. This has the advantage of making the structure of the logic of a program clear and high-level. Simply put, it makes the alternating structure of a logic program easier to grasp. In classical logic programming languages, logical loops must be expressed by means of recursion using both multiple procedure definitions and cut. In 2LP logical loops and logical alternation can be expressed in a structured way. By way of example the following 2LP code uses an **`or`** loop nested in an **`and`** loop to generate permutations of **`N`** symbols in lexicographical order. It is a variation of the classical algorithm given, for example, in [Reingold *et al.*]. The **`and`**-**`or`** construct can often be used to enumerate the possibilities for a search problem. It is the **`and`** that moves the search forward to the next level. When a failure occurs, the **`or`** has kept track of where to start trying next. The loop control variables are handled declaratively in 2LP in that their proper values are restored upon backtracking. In the first bit of code below, failure is triggered by a **`fail`** statement, but in the scheduling example that follows, it is when a potential schedule cannot satisfy the constraints, that failure occurs and the next permutation is tried. This method of generating permutations is useful in 2LP because the lexicographic order meshes correctly with backtracking mechanism of the logic programming search.

```
#define N ...
int permu[N];
int p[N][N];
2lp_main()
{
      and(int i=0;i<N;i++) p[0][i] = i; //The identity permutation
```

```
	and(int i=0;i<N;i++)
		or(int j=0;j<N-i;j++){
			if j == 0; then and(int k=i+1;k<N;k++) p[i+1][k] = p[i][k];
			permu[i] = p[i][i+j];
			if j != 0; then p[i+1][i+j] = p[i][i+j-1];
		}
	and(int i=0;i<N;i++) printf("%d ",permu[i]);printf("\n");
	fail; //forces backtracking to generate next permutation
}
```

The **and-or** construction is natural to use as a form of ∀∃. In this way, in working with NP-problems, it can play the role of the non-deterministic guess in the Garey-Johnson model [Garey,Johnson]. In the context of Complexity Theory, the non-determinism is factored out as one sequence of guesses; in the context of actual programming, candidate guesses are generated incrementally and so take an **and-or** or ∀∃ structure.

In 2LP procedural techniques for doing recursion and for restoring state in a search application are often used in conjunction with the built-in logic programming machinery which restores continuous variables and loop control variables. Let us look an example which uses the lexicographic ordering on permutations. Schedules which require permuting a sequence of tasks in order to meet delivery, production and other kinds of constraints are called *permutation schedules*. The following code segment is taken from an application to a permutation scheduling problem. The continuous variable **start_time[i]** represents the starting time of task **i**, **permu[k]** is the task slated for the **k**th position in the schedule, **prod[i]** is the production time required for task **i** and **dt[i]** is the time by which task **i** must be finished.

```
#define N ... // number of tasks to schedule
extern double prod[], //prod[i] is time required for task i
                dt[]; // dt[i] is delivery time for task i
int permu[N];
int p[N][N];
continuous start_time[N];
            //start_time[i] is the start time for the ith task

2lp_main(){
	//declarative knowledge
	//priority constraints such as
	start_time[i] >= start_time[j] + prod[j]; //task i depends on task j
	    ...
	//delivery constraints such as
	start_time[i] + prod[i] <= dt[i]; //task i must be finished by time dt[i]
	      ...
	//throughput constraints such as average job must be finished
	//within 2/3 of the total scheduled time
	(1.0/N)*sigma(int i=0;i<N;i++)(start_time[i] + prod[i]) <= .
			.66 * sigma(int i=0;i<N;i++) prod[i];
	      ...
	//procedural part of program
	and(int i=0;i<N;i++) p[0][i] = i;//initialize identity permutation
	if schedule_all();
	then and(int i=0;i<N;i++)
		printf("Schedule task %d in the %dth position\n",permu[i],i);
	else printf("There is no feasible schedule\n");
}
```

```
schedule_all() {
      and(int i=0;i<N;i++)
            or(int j=0;j<N-i;j++) {
                  if j == 0; then and(int k=i+1;k<N;k++) p[i+1][k] = p[i][k];
                  //schedule task p[i][i+j]
                  start_time[p[i][i+j]] ==
                        sigma(int k=0;k<i;k++) prod[permu[k]];
                        // fails if inconsistent with setup constraints
                        // and those generated thus far by this statement
                  permu[i] = p[i][i+j];
                  if j != 0; then p[i+1][i+j] = p[i][i+j-1];
            }
}
```

4. Factoring the logic in and out

As a programming language for classical MIP applications, the fact that 2LP is a logic based language is a significant advantage. For it can handle logical decisions either by means of additional 0-1 variables or by direct encoding of the logic using the 2LP program constructs. Thus, for instance, 2LP can make full use of an important idea that comes from MIP modeling, namely the use of continuous variables as fuzzy logical variables. For example, if the program contains several "constants" such as a fixed-cost or capacity whose values are in fact functions of a single logical decision, then these values can be parametrized as (linear) functions of a continuous variable; this variable is then bound by 0 and 1 and is made equal to 0 or 1 depending on the logical decision made. This yields a fuzzy or continuous declarative approximation to the model sought and can be very useful both from a heuristic point of view and a pruning point of view. However, MIP modeling is not based on a logic language and logical disjunction and other connectives can only be expressed by the introduction of additional 0-1 variables. These variables are used to encode switches that express the logical relationships defined by the application. On the negative side, the introduction of these variables can greatly swell the total number of variables of the model and can add a considerable number of constraints. (This means adding rows and columns to the matrix sent to the linear programming solver; this is exacerbated by the fact that a row also will add an additional column for its slack variable. This comes down to a quadratic increase in the size of this matrix.) With the logical control that is built in to 2LP, coding logic in terms of 0-1 variables can most often be eliminated; the result can be both elegance in modeling and speed in performance. However, in other cases, additional variables can serve to "tighten" a model and to make the "linear relaxation" fit the disjunctive model more closely. This closeness of fit can be most important in the phase of an optimization program where the job remaining is to verify that the current best solution cannot be improved. 2LP facilitates the development of both loose and tight models and the transition back and forth between them. For examples, we refer the reader to [McAloon,Tretkoff 3].

5. An Example: Goal Programming with Logic

The term *goal programming* is used to describe situations where there are several criteria that must be taken into account to have the best solution to a problem. Goal programming is usually done only for linear programming models. In the application that follows we consider a goal programming case where logic, in the form of integrality requirements, is needed in addition to the straight linear model.

In this application the task is to schedule toll takers. An analysis of traffic patterns has determined the number of employees required for each hourly period during the 24 hour day. Each worker comes on for a 9 hour shift with a 1 hour break in the middle. Management wants to minimize the total number of workers assigned each day. Thus the program must determine the number of workers who should start their shifts at each hour of the day so as to minimize the total number of workers hired. The model requires, of course, a solution which determines an integral number of workers to start at each hour, and it is this requirement that necessitates a logical search strategy. Unavoidably, there will be periods where the number of workers available exceeds the number required for that period. These people will of course be able to take care of other duties beside collecting tolls. It is desirable to find a schedule which distributes these extra workers as evenly as possible during the day so that continuity can be maintained in these auxiliary jobs.

The first task of the model will be to determine the minimum number of workers required. This will be done with a first objective function. Then, once the minimal total number of workers is found, the next task is to distribute the workers who will not be collecting tolls as evenly as possible during the 24 hour day. For that two stratagems will be employed in succession. First, the largest number of extra workers that can occur at any one hour will be minimized and then the number of different time periods that can have extra workers will be maximized. To do all this, for the first two phases of the program, i.e., determining the minimal number of workers needed and then minimizing the largest number of extra workers at any hour, the built-in optimizer construct **`min:`** ... **`provided {}`** is used; with this construct, after the optimum is determined the objective function for that optimization call is set equal to its optimal value but otherwise the continuous variables of the program are only constrained by the constraints that were active before the optimizing call was made. This construct supports a "bluff": since the number of workers is necessarily an integer, to avoid alternative solutions which determine the same number of workers, each successive solution found during a search phase of the problem should be required to better the previous solution by at least 1. The command **`bluff(1.0)`** sets this machinery in motion. (The term "cheat" is also used in this context in the literature, and "bluff" is sometimes used to denote an initial upper bound on the objective function in a minimization problem or lower bound in a maximization problem.) For the final task, the **`max:`** ... **`subject_to {}`** construct is used; upon completion of this block the continuous variables are fixed to the values the witness point had when the optimal solution was found.

The only data in the model is the number **`d[j]`** of toll takers required for each hourly period.; this data is obtained as an external array of integers. The only constraints needed are the ones that sum up the number of toll takers on duty for each hourly period **`j`**. We will let the continuous variable **`x[j]`** represent the number of workers who come on at the **`j`**th hour. Looking ahead to hour **`j+8`**, the workers on duty will be all those who came on at some time **`j+i`** for **`i=0,...,3`** and **`i=5,...,8`**. For each **`j+8`** this requirement can be expressed as

```
sigma(int i=j;i<=j+8;i++)x[i%N] - x[(j+4)%N] <= d[(j+8)%N]
```

where **`N`** is a defined constant equal to 24. Naturally, we have to subtract the term **`x[(j+4)%N]`** which represents the workers on break. If we let **`sur[i]`** be the number of surplus workers at hour **`i`**, a variable which will be used in the second and third phases of the solution process, this can be written

```
sigma(int i=j;i<=j+8;i++)x[i%N] - x[(j+4)%N] - sur[(j+8)%N] == d[(j+8)%N]
```

In each of the three phases of the program, in order to meet the integrality requirements on the **`x[i]`**, the model employs a "lazy" strategy which is akin to a priority argument in Recursion Theory. The linear relaxation is optimized and there is a circular loop around the 24 time periods. If **`wp(x[i])`**, the **`x[i]`**-coordinate of the witness point, is currently at an integer value, this time slot is skipped over and the loop continues until a variable is encountered such that the witness point's coordinate at that variable is not integral; then the process is begun to drive that variable to an integral value. To describe this process, let **`t`** be equal to **`floor(wp(x[i]))`**; then the search loops through the constraints **`t == x[i]`**, **`t-1 == x[i]`**,... continuing upon backtracking toward **`0 == x[i]`**. A fact about convexity is brought into play - if fixing **`x[i]`** at a an integer value **`t-k`** is linearly infeasible, then the search in that direction can be cut off since no smaller value will be linearly feasible. After the search in the downward direction, the search proceeds upward from **`t+1`**. Naturally, a variable which was at an integer position might be "injured" in the branching and optimization process required to make another variable have an integral value. Either this variable will return to an integral position or it will eventually be reached in the circular loop and branched upon. Thus all integrality "requirements" are met.

```
#define N 24 // hours in the day
#define M 8 // 8 working hours

continuous x[N]; // x[i] is number of workers starting at hour i
```

```
continuous sur[N]; // sur[i] is number of surplus workers at hour i
continuous z,u, ex[N],tra[N]; // auxiliary variables

int s; // circular loop counter
extern double d[]; // data entered in external C code

2lp_main()
{
      sigma(int i=0;i<N;i++) x[i] == z; //z is to be minimized

      set_up_constraints();

      schedule1();
      schedule2();
      schedule3();

printf("The total number of workers required is %d\n",nint(z));
printf("There will be surplus workers during %d hourly periods\n\n",nint(sig-
ma(int k=0;k<N;k++) tra[k]));
printf("The schedule recommended is \n\n");
and(int i = 0;i<N;i++)
      printf("At hour %i \t bring on %d workers\n",i,nint(x[i]));
}

schedule1(){ // determine minimum number of toll takers required
      min: z; provided {//only restores optimal value of objective function
            bluff(1.0);
            optimize();
            s = N-1;
            lazy_loop();
      }
}

schedule2(){ // minimize the maximum surplus workers
      and(i=0;i<N;i++) sur[i] <= u;
      min: u; provided {
            bluff(1.0);
            optimize();
            s = N-1;
            lazy_loop();// find minimum hourly surplus bound
      }
}

schedule3(){ // distribute surplus workers more broadly

//cap on number of surplus workers at each hour
      and(i=0;i<N;i++) sur[i] <= u;

//tra[i] is "characteristic function" for presence of surplus workers
      and(i=0;i<N;i++) tra[i] <= 1;

// decompose hourly surplus
      and(int k=0;k<N;k++) sur[k] == ex[k] + tra[k];
```

```
// favor wide distribution of surplus workers
      max: sigma(int k=0;k<N;k++) tra[k]; subject_to {
                  bluff(1.0);
                  optimize();
                  s = N-1;
                  lazy_loop();
      } //subject_to restores value at optimum of
        //all continuous variables

}


lazy_loop(){
      and(int i=0;i<N;i=(i+1)%N)
            if !integral(x[i]); // fabs(wp(x[i]) - nint(x[i])) > EPSILON
                  then down_and_up(i,floor(x[i]),x[i]);
            else if i == s; then break; //break exits and-loop with success
}

down_and_up(int i,double t,continuous x){//i,t passed by value, x by reference
      either // down
            or(int k=t;k>=0;k--)
                  if x == k; then {optimize(); s = i;}
                  else break;
      or // up
            or(int k=t+1;k<=M;k++)
                  if x == k; then {optimize(); s = i;}
                  else break; //break exits or-loop with fail
}

set_up_constraints(){

      and(int j=0;j<N;j++)
            sigma(int i=j;i<=j+M;i++) x[i%N] - x[(j+4)%N] //9 hours less break
                        - sur[(j+M)%N] == d[(j+M)%N];
}
```

6. Future directions

An interesting topic combining logic and constraints is propositional theorem proving. Experimental work in this field using polyhedral methods is being carried out in several research centers in the USA and abroad, e.g. [Hooker],[Comon *et a*l.], [Hahnle], [Bell *et al*.]. Our starting point is the analysis of [Blair,Jeroslow,Lowe] and Jeroslow's watershed monograph [Jeroslow] where it is argued that Davis-Putnam-Loveland (DPL) techniques are more amenable to constraint methods than resolution based techniques. The relation of propositional consequence is co-NP complete. With resolution methods, a problem to show there is no polynomially long path of a certain type is transformed into an existential search for a successful path of possibly exponential length. The proper analogy from Recursion Theory is the equivalence of Π^1_1 sets of integers and ω_1^{CK} -recursive sets of integers. (Note that in this view NP is analogous to co-RE and the analytic sets of recursion theory and descriptive set theory.) The DPL approach keeps the consequence relation in its co-NP form and makes it akin to a MIP optimization problem. We have implemented a version of DPL in C calling 2LP to handle the constraints. The first difference with symbolic

DPL methods is that the unit resolution check for consistency is done automatically by the simplex based linear programming solver. Note that this also means that the linear programming solver provides a complete theorem prover for the Horn clause fragment of the set of clauses. Using the 2LP programming language tool we will also be able to add to the continuous mathematical version of DPL heuristics and search techniques which exploit the geometry of the polyhedral set defined by the propositions interpreted as constraints. This puts at our disposal, heuristics such as the Balas-Martin heuristic from Integer Programming, greedy and Gray code methods, the Jeroslow heuristic for DPL as well as some new opportunistic methods for driving continuous variables to discrete values that we have developed in our own work.

Among important non-traditional connectives are generalized disjunctions such as 5 choose 3. When expressed in clause form these connectives are not very amenable to unit resolution and similar symbolic techniques. However, it can be shown that in conjunction with constraint methods, these generalized disjunctions can be used to detect early failure in the search for a satisfying assignment to a set of propositions. Let us look at a concrete example. The generalized disjunction "At most one of p,q,r is true" transforms into three clauses, namely, {p',q'},{q',r'},{p',r'}. If we add to these the clauses {p,q},{p',q},{p,q'},{q,r} the resulting set is inconsistent. However, unit resolution cannot detect this inconsistency and the symbolic DPL algorithm can not detect it without resorting to the splitting rule. As linear constraints, these clauses become

$$q + r >= 1;\ (1-p) + (1-q) >= 1;\ (1-q) + (1-r) >= 1;\ (1-p) + (1-r) >= 1;$$

$$p + q >= 1;\ (1-p) + q >= 1;\ p + (1-q) >= 1;\ 0 <= p,q,r <= 1;$$

This set of constraints is still consistent as a set of linear constraints; in fact, by Chapter 5 of [Jeroslow], a set of clauses with no unit clauses is always consistent from the linear point of view. But, if we also add the generalized disjunction "At most one of p,q,r is true" in the form $p + q + r <= 1$, the linear constraints become inconsistent.

2LP has been designed with the purpose of bringing into software control and search mechanisms which in MIP and other constraint systems are either hidden in a black-box or simply not available. This aim is being realized to a definite extent and promises to continue to extend its range. The next major challenge is to bring into the 2LP orbit (or that of another constraint language) the techniques of "polyhedral theory" and "branch 'n' cut." Certainly the work on MINTO [Savelsbergh,Sigismondi,Nemhauser] makes one believe that this can be achieved with some success. In our opinion this will prove important both for optimization applications and constraint-based reasoning systems and theorem-provers. The reason is that "branch 'n' cut" methods encode a kind of "lemma" in the form of constraints which must hold at any solution below the current node in the search. Adding the constraint description of "At most one of p,q,r is true" to the clause description in the example above is a "lemma" in a similar way. These "lemmas" are special in that unlike an ordinary lemma, they become "resident" and stay in force for the rest of the computation once they are added to the constraint set. In traditional automated reasoning systems, in rewrite-based symbolic computation systems, and in ordinary human mathematical theorem proving, a search process must be invoked to find a previously established lemma and to apply it locally. The "resident lemmas" can be applied once and for all with no additional invocation required. As far we know, this technique has not been applied in expert systems or symbolic computation systems.

Let us make a "remark" in the mathematical sense: for simplicity, consider the situation where a model has no solution. (This is the case of an optimization model once the best possible solution has been found and it is the situation in a DPL based propositional theorem prover if the theorem to be proved is true!) For simplicity also suppose that a state or node in the search can only be reached one way in either depth-first or breadth-first search and finally suppose that the total number of nodes is finite. Then (it can be seen) both depth-first and breadth-first search visit all possible nodes. It follows from this remark that for all practical purposes in the above situation, exhaustive enumeration by depth first search is the most efficient means of verifying that there is no better solution among classical methods such as A*, Land and Doig etc. This is not a particularly pleasant prospect since it is so limiting; "polyhedral theory" and "branch 'n' cut" are the best candidates to look to at this time for some improvement. Parallel processing is another candidate; in [Atay] arguments are given as to why a parallel search will not visit unnecessary or duplicate nodes in this situation.

Another promising application of constraint programming is to respond to the challenge launched by [Dhar,-Ranganathan] where it is argued that rule based expert systems are superior to mathematical programming methods for attacking complex constraint problems; in brief they contend that MIP methods do not allow for local control of decisions and do not provide a mechanism for providing good partial solutions.

7. Concluding remarks

The 2LP system currently runs on UNIX workstations and 386/486 PCs. A parallel implementation which basically is an or-parallelization of the depth-first 2LP stack mechanism has been done. This work is reported on in [Atay,McAloon,Tretkoff].

The authors would like to acknowledge the support of NSF grant CCR-9115603.

8. Bibliography

[Atay] C. Atay, Parallelization of the Constraint Logic Programming Language 2LP, Ph.D. Thesis, City University of New York, June 1992

[Atay,McAloon,Tretkoff] C. Atay, K. McAloon and C. Tretkoff, 2LP: a highly parallel constraint logic programming language, *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, March 1993

[Bell *et al.*] C. Bell, A. Nerode, R. Ng and V. S. Subrahmanian, Implementing deductive databases by linear programming, University of Maryland, Computer Science Technical Report, CS-TR-2747, UMIACS-TR-91-122.

[Blair,Jeroslow,Lowe] C. W. Blair, R. G. Jeroslow, J. K. Lowe, Some results and experiments on programming techniques for propositional logic, *Computer and Operations Research* **13**(1986) 633-645.

[Colmerauer] A. Colmerauer, An introduction to Prolog III, *CACM* **33** No. 7 (1990) 69-91.

[Comon *et al.*] H. Comon, H. Ganzinger, C. Kirchner, H. Kirchner, J.-L. Lassez, G. Smolka (editors): Theorem Proving and Logic Programming with Constraints, Dagstuhl-Seminar-Report; 24 (9143)

[Cox,McAloon,Tretkoff] J. Cox, K. McAloon, C. Tretkoff, Computational complexity and constraint logic programming, *Annals of Mathematics and Artificial Intelligence*, **5**(1992) 163-190.

[Dhar,Ranganathan] V. Dhar and N. Ranganathan, Integer Programming vs. Expert Systems: an experimental comparison, *Communications of the ACM* **33**, No. 3 (1990) 323-337.

[Dincbas et al.] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, F. Berthier, The Constraint Logic Programming Language CHIP, *Proceedings of the International Conference on Fifth Generation Computing Systems*, 1988.

[Fourer,Gay,Kernighan] R. Fourer, D. Gay and B. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, The Scientific Press, 1993

[Garey,Johnson] M. Garey and D. Johnson, *Computers and Intractability*, W.H. Freeman 1979

[Hahnle] R. Hahnle, A new translation for deduction into integer programming, submitted for publication.

[Hooker] J.N. Hooker, Jr., Input proofs and rank one cutting planes, *ORSA Journal on Computing* **1**, No. 3(1989) 137-145.

[Jaffar,Michaylov] J. Jaffar, S. Michaylov, Methodology and Implementation of a CLP System, *Proceedings of the 1987 International Logic Programming Conference,* edited by J.-L. Lassez*,* MIT Press, 1987.

[Jeroslow] R. G. Jeroslow, Logic-Based Decision Support, Mixed Integer Model Formulation, North-Holland, New York, 1989.

[Jeroslow,Wang] R. G. Jeroslow, J. Wang, Dynamic programming, integral polyhedra and Horn clause knowledge bases, *ORSA Journal of Computing* **1**, No. 1(1989) 7-19.

[Lassez] J.-L. Lassez, From logic programming to lazy programming, manuscript.

[McAloon, Tretkoff 1] K. McAloon and C. Tretkoff, Subrecursive constraint logic programming, *Proceedings of the NACLP 1990 Workshop on Logic Programming Architectures and Implementati*on*,* edited by J. Mills.

[McAloon,Tretkoff 2] K. McAloon and C. Tretkoff, *Logic and Optimization*, in preparation

[McAloon,Tretkoff 3] K. McAloon and C. Tretkoff, AI/OR Modeling in 2LP, Brooklyn College Computer Science Technical Report 93-4.

[Reingold *et al.*] E. Reingold, J. Nievergeld, and N. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, In., Englewood Cliffs, NJ, 1977

[Savelsbergh,Sigismondi,Nemhauser] M. Savelsbergh, G. Sigismondi and G. Nemhauser, Functional description of MINTO, a MIxed INTeger Optimizer, Georgia Institute of Technology Tech Report 30332, May 1992.

[Warren] D.H.D. Warren, An Abstract Prolog Instruction Set, Technical Report 309, SRI International, 1983.