Logic, Modeling, and Programming

KEN MCALOON AND CAROL TRETKOFF

Logic Based Systems Lab
CUNY Graduate Center and Brooklyn College
2900 Bedford Avenue
Brooklyn, NY 11210
E-mail: mcaloon@sci.brooklyn.cuny.edu
E-mail: tretkoff@sci.brooklyn.cuny.edu

In this paper we discuss the integration of logic, modeling, and programming in order to solve problems in operations research, artificial intelligence, and decision support programming in general. Our goals are to integrate modeling into the larger programming scheme of things and, conversely, to inject programming into modeling. To accomplish these ends, we use the language 2LP, which is based on ideas from constraint logic programming. This leads to a technologically open way to handle problems, one which supports flexible treatment of goal programming, hybrid MIP/local search algorithms, libraries for distributed processing, disjunctive programming, etc. An additional advantage of the programming language approach is that problem solving and model management can be abetted by software engineering techniques. In this paper, by means of variations on a single example, we will illustrate how the logical connectives and linear constraints interact in the solution of a linear program, a goal program, a disjunctive program, a branch and bound search, a randomized shuffle algorithm, and a parallel solution to a model with stochastic data.

1 Integrating Modeling and Programming

The theme of this paper is the integration of logic, modeling, and programming. In particular, the goal is to integrate modeling into the larger programming scheme of things and, conversely, to inject programming into modeling. Historically, modeling has been done by means of matrix generators and declarative algebraic languages. These languages facilitate the formulation of a linear or mixed integer linear programming problem in concise terms, and the emphasis is on the declarative description of the application. Here, we integrate declarative modeling, programming with logic, and the elements of procedural programming.

We will carry out the goal of integrating logic, modeling, and programming using the 2LP language. Emphasis will be placed on the logical connectives and how they enable the expression of both the modeling of a problem and the control over its solution. The 2LP language provides the programming support needed for linear constraints, for the backtracking of logic programming, and for the standard procedural programming techniques of abstraction and encapsulation. The handling of the logical connectives is important because constraints, unlike assignment statements, can fail, and there must be constructs in the language that enable the program to detect infeasibility and continue down a different path if necessary. For this, we use the technology of constraint logic programming [12], [23].

The 2LP embedding of linear programming practice into a programming language environment, in fact, embeds the larger disciplines of mixed integer programming and disjunctive programming, which is an extension of mixed integer programming, [8], [9], [19]. The language features also facilitate extensions of linear programming to goal programming, stochastic programming, "what-if" analysis, etc. In addition, 2LP supports the integration of MIP and disjunctive programming techniques with randomized local search algorithms.

These topics will be developed through a single example, the multi-period planning problem of [37]. We will make a journey through the four basic logical connectives, showing with examples how each connective is associated with a new range of problem solving ability. We will start with a straightforward version of the planning problem using only conjunction and standard programming practice. We will continue with a goal programming version that uses the form of disjunction commonly employed in porgramming languages. Next, we will show how to handle the problem when logical requirements are added; the backtracking version of disjunction will be used here. Once negation has been added to our repertoire, heuristic search with lookahead will be employed. We also discuss a solution of the blending example using branch-and-bound search and randomized local search. Finally, we comment on a parallel solution to a stochastic variant of the problem.

2 The 2LP Programming Language

The 2LP language is a constraint logic programming language with C like syntax. The data type `continuous` is introduced in order to capture the modeling methods of linear programming. Constraints on continuous variables define a polyhedral set called the *feasible region*. The representation of this convex body (by means of the simplex method) includes a distinguished vertex, called the *witness point*. Typically at the end of a program, this vertex will be the solution to an optimization problem. 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 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. However, in order to support full algebraic modeling and to support the logical connectives in a structured way, the language approach is the most natural one. For the other approach to linear programming and constraint programming, see [31] and [32].

2LP is different from algebraic modeling languages such as GAMS [11] and AMPL [17] in several ways. First, it is a logical control language as well as a modeling language and so supports search methods not usually available to the MIP modeler. Second, it can directly code disjunctive programs and avoid recoding as a MIP model. Third, 2LP supports parameter passing, procedure based modularity and other structured programming constructs. Fourth, the data types `int` and `double` in 2LP are the same as in C/C++ and so the 2LP model can be passed pointers to data directly from C/C++ avoiding a detour through a data file.

As a programming language, 2LP is designed to be embedded in the larger programming scheme of things. The 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. 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.

3 Conjunction and Structured Linear Programming

In symbolic logic, the fundamental properties of conjunction are commutativity and idempotence. With the shift from mathematics to programming, procedural elements such as change of state are introduced and the semantics of conjunction changes. For example, in C/C++, conjunction is neither idempotent nor commutative because of the possibility of side effects. Constraints present an opportunity to recapture some of the mathematical semantics of conjunction.

For classical linear programming the critical logical connective is conjunction, because a linear program consists of a conjunction of constraints. In 2LP conjunction is expressed by juxtaposition, as in most all programming languages, and by means of the `and` loop construct. To start, let us code the standard linear program: $\mathbf{x} \geq \mathbf{0}$, $\mathbf{Ax} \leq \mathbf{b}$, $z = \mathbf{cx}$, $max\ z$. The constraints have the form

$$\sum a_{ij} x_j \leq b_i$$

and the objective function is

$$z = \sum b_j x_j$$

This translates almost literally into 2LP code. The non-negativity constraints on the continuous variables are assumed by default. An `and` loop is used to run through the multivariable contraints; each such constraint is written with the `sigma` loop which is expresses sums as in the usual mathematical notation. For concreteness, some specific data are included in the following example which is a complete program.

```
#define M 3        // Number of constraints
#define N 4        // Number of variables

2lp_main()
{
double a[M][N],b[M],c[N];  // A, b and c
continuous X[N],Z;         // The continuous variables

      // Array assignments, same as initialization in C/C++
      a = {
            1.1, 2,2, 3,3, 4.4,
            5.6, 4.8, 5.9, 6.5,
            11.1, 21.0, 13.0, 6.0
      };
      b = { 3.1, 4,2, 9.8 };
      c = { 121.4 , 333.6 , 234.0 };

      // The constraints Ax <= b
      and(int i=0;i<M;i++)
            sigma(int j=0;j<N;j++) a[i][j]*X[j] <= b[i];

      // The objective function
      Z == sigma(int j=0;j<N;j++) c[j]*X[j];

      max: Z;     // Optimize Z

      printf("Maximum value of the objective function is %f\n",Z);
}
```

Note that the equality constraint defining `Z` is written with the relational operator `==` while the `=` operator is used for assignment.

To illustrate these features further we will work with the multiperiod blending problem of [37]. A company produces a cooking oil. This oil is made by blending

Table 1
The prices for the next six months

	Corn	Peanut	Safflower	Sesame	Sunflower
Jan	115	110	130	120	110
Feb	115	90	110	130	130
Mar	95	100	130	140	110
Apr	125	120	120	110	120
May	105	110	150	120	100
June	135	80	140	100	90

Table 2
The hardness coefficients

Corn	5.0	Peanut	4.2	Safflower	2.0
Sesame	6.1	Sunflower	8.8		

5 different vegetable oils. The unprocessed oils are purchased from distilleries. Purchase of the raw ingredients and production of the finished product are organized on a monthly basis and management would like to formulate a 6 month plan to optimize revenue. The following table gives the posted price per ton of each vegetable oil for the next 6 months. These prices can be locked in now by means of futures contracts and are given in Table 1.

The cooking oil that is produced during a given month can be sold during that month and brings in £150 per ton. The company's process is efficient and for all practical purposes the amount of oil produced is equal to the sum of the amounts of the vegetable oils used.

Vegetable oil that is in stock but not used during a given month can be kept in storage and used in the following months. In fact, at the start of this 6 month period, there are 500 tons of each kind of vegetable oil in storage and the plan should provide for the same quantities of oil in storage at the end of the 6 months. The storage capacity for each vegetable oil is 1000 tons; the cost of keeping oil in storage is assessed at the rate of £5 per ton at the beginning of each month.

Processing the cooking oil is subject to various requirements. Each month there is a limit of 250 tons on the total amount of corn, peanut, and safflower oil that can be processed and a limit of 200 tons on the total amount of sesame and sunflower oil that can be processed. The blended cooking oil must also meet some hardness requirements. Each vegetable oil has a hardness coefficient which is a number between 1.0 and 10.0. The weighted average of hardness of the oils in the blend must be between 3.0 and 6.0. These coefficients are given in Table 2.

The object is to formulate a 6 month plan which will specify the amount of each vegetable oil to purchase each month, the amount of cooking oil to produce each month, and the amount of each vegetable oil to hold over in storage for the succeeding month.

Let us begin the design of this model by considering the data given in Table 1. In the code, we will use the symbolic constants `MONTHS` and `KINDS` to represent the number of months, 6, and the different kinds of oil, 5. The data of Tables 1 and 2 can then be stored in the arrays `price[MONTHS][KINDS]` and `hardness[KINDS]`, both of type `double`. Other symbolic constants are also useful: in the program to handle the particulars of the problem.

```
#define MONTHS 6          // Number of periods
#define KINDS 5           // Kinds of vegetable oils
#define TONPRICE 150      // Selling price per ton
#define STORCOST 5        // Storage cost per ton
#define INITSTOR 500      // Initial storage in tons
#define ENDSTOR 500       // Final storage in tons
#define STORCAP 1000      // Monthly storage capacity

double price[MONTHS][KINDS];
double hardness[KINDS];
```

Now let us turn to the quantities that have to be determined. There are limits on how the oils can be combined. We will define `LIMITCPS` as 250 and `LIMITSS` as 200 to represent these data values.

The blended cooking oil must also meet the hardness requirements in the array `hardness[KINDS]` of type `double`. The bounds on average hardness are `LBHARD` and `UBHARD`. This leads to some further definitions.

```
#define LIMITCPS 250       // Limit on corn, peanut, and
                           // safflower production
#define CPS 3              // Number of oils in this group
#define LIMITSS 200        // Limit on sesame and sunflower
                           // production
#define LBHARD 3.0         // Lower bound on average hardness
#define UBHARD 6.0         // Upper bound on average hardness
```

Reiterating, the objective is to formulate a 6 month plan which will specify the amount of each raw oil to purchase each month, the amount of cooking oil to produce each month, and the amount of each unprocessed oil to hold over in storage for the succeeding month. We can encapsulate this in an array `Plan[MONTHS][MODES][KINDS]` of type `continuous`, where `MODES` is defined as 3. For improved readability of the code, we will also define `PURC`, `PROC`, and `STOR` to represent the three modes: purchased, processed and stored:

```
#define MODES 3       // Purchase, Process or Store
#define PURC 0
#define PROC 1
#define STOR 2
```

We will use a continuous variable `Revenue` to represent the revenue generated by the sale of the finished products. So the declarations of the continuous variables are

```
continuous Plan[MONTHS][MODES][KINDS];
continuous Revenue;
```

Continuing to approach this application in a top-down manner, we can formulate an outline for the program as follows:

1. Fill the price and hardness arrays
2. Establish constraints on the initial and final storage amounts
3. Constrain the monthly storage capacities
4. Constrain the hardness and production limits of the blend for each month
5. Link the months so that oils stored last month may be used this month
6. Write an equation for the revenue
7. Maximize the revenue
8. Then print the plan

The first six tasks set up the constraints of the problem. The last two carry out the optimization and produce a report. Using procedures to carry out these tasks, the first 6 translate into:

```
setup()                 // Points 1-6
{
      data(price,hardness);                          // 1

      first_stor(Plan[0]);                           // 2
      last_stor(Plan[MONTHS-1][STOR]);

      and(int i=0;i<MONTHS;i++){
            storage_caps(Plan[i][STOR]);             // 3
            blend_constraints(Plan[i][PROC]);        // 4
      }

      and(int i=1;i<MONTHS;i++)
            link(Plan[i-1][STOR],Plan[i][PURC],
```

```
                Plan[i][PROC],Plan[i][STOR]);      // 5

      objective_function(Plan,Revenue,price);      // 6
}
```

The main routine will call `setup` and then proceed to points 7 and 8.

```
2lp_main()
{
      setup();                    // 1 - 6
      max: Revenue;               // 7
      output(Revenue,Plan);       // 8
}
```

Let us now develop the procedures for the setup. First, `data` merely fills its two array parameters with the data from the corresponding tables.

```
data(double price[][KINDS],double hardness[])
{
      // Array assignment - same as initialization in C/C++
      price = {
                115, 110, 130, 120, 110,
                115,  90, 110, 130, 130,
                 95, 100, 130, 140, 110,
                125, 120, 120, 110, 120,
                105, 110, 150, 120, 100,
                135,  80, 140, 100,  90
      };

      hardness = { 5.0, 4.2, 2.0, 6.1, 8.8 };
}
```

The procedure `first_stor` needs to set up the equality for the storage at the end of the first month for each kind of vegetable oil. This will be the initial storage of 500 (or `INITSTOR`), adjusted by the amounts purchased and processed during the first month. This gives us

```
first_stor(continuous FirstMonth[][KINDS])
{
      and(int j=0;j<KINDS;j++)
              FirstMonth[STOR][j] ==
                      INITSTOR - FirstMonth[PROC][j] +
                                       FirstMonth[PURC][j];
}
```

Since we need to refer only to the first month of the Plan array, we will pass only the slice Plan[0] to first_stor, and not the entire array. In essence, the procedure treats FirstMonth as a two-dimensional array.

The procedure last_stor is simpler. The amount of each oil stored at the end of the last month must be bound to 500, or ENDSTOR. Only the vector of oils to be stored in the last month, Plan[MONTHS-1][STOR], will be passed to the procedure from the Plan array.

```
last_stor(continuous LastMonth[])
{
      and(int j=0;j<KINDS;j++)
            LastMonth[j] == ENDSTOR;
}
```

The storage_caps procedure will bound the storage each month by at most 1000 tons, or STORCAP. The program calls this procedure for each month, and passes only the vector of the Plan array that refers to that month's storage, namely Plan[i][STOR]. Each of the elements of this vector, called Stored in this procedure, should be bound by STORCAP as follows.

```
storage_caps(continuous Stored[])
{
      and(int j=0;j<KINDS;j++)
            Stored[j] <= STORCAP;
}
```

Let us now turn to the production constraints which involve oils of different kinds processed together. These constraints are to be stated in a procedure which requires the vector of the oils to be processed in a given month, so it is called for each month i by the statement blend_constraints([Plan[i][PROC]). The procedure will have the signature

```
blend_constraints(continuous Oils[KINDS])
```

where the identifier Oils refers to the vector of this month's processed oils.

Let us develop the blend_constraints routine. After the bounding constraints on the individual variables, the next two constraints bound the combined amounts of each subgroup of oils by LIMITCPS and LIMITSS. Since we have enumerated the subgroup of corn, peanut, and safflower oils as 0, 1, and 2, and the other subgroup of sesame and sunflower oils as 3 and 4, we can write these constraints using two loops:

```
      sigma(int j=0;j<CPS;j++) Oils[j] <= LIMITCPS;
      sigma(int j=CPS;j<KINDS;j++) Oils[j] <= LIMITSS;
```

We now need to write the product's hardness constraint. This is a ratio constraint which we formulate as,

$$\sum(\text{hardness}_i \times \text{Oils}_i) \geq \text{LBHARD} \times \sum \text{Oils}_i$$

or, in code, as

```
      sigma(int i=0;i<KINDS;i++) hardness[i]*Oils[i]
                     >= LBHARD*sigma(int j=0;j<KINDS;j++) Oils[j];
```

Similarly, the constraint for the upper bound of `UBHARD` can be written as

```
      sigma(int i=0;i<KINDS;i++) hardness[i]*Oils[i]
                    <= UBHARD*sigma(int j=0;j<KINDS;j++) Oils[j];
```

We can now aggregate these statements in the procedure

```
blend_constraints(continuous Oils[KINDS])
{
      // Upper bounds on variables
      and(int j=0;j<CPS;j++) Oils[j] <= LIMITCPS;
      and(int j=CPS;j<KINDS;j++) Oils[j] <= LIMITSS;

      // Global limit on CPS oils
      sigma(int j=0;j<CPS;j++) Oils[j] <= LIMITCPS;

      // Global limit on SS oils
      sigma(int j=CPS;j<KINDS;j++) Oils[j] <= LIMITSS;

      // Lower hardness bound
      sigma(int j=0;j<KINDS;j++) hardness[j]*Oils[j]
            >= LBHARD*sigma(int j=0;j<KINDS;j++) Oils[j];

      // Upper hardness bound
      sigma(int j=0;j<KINDS;j++) hardness[j]*Oils[j]
            <= UBHARD*sigma(int j=0;j<KINDS;j++) Oils[j];
}
```

This brings us to the linking procedure whose signature will be

```
link(continuous OldSt[KINDS],Purc[KINDS],Proc[KINDS],NewSt[KINDS])
```

The parameters are vectors that represent the amount of each kind of oil which has been held over in storage from the previous month, and the amounts which are to be purchased, processed and stored this month, respectively. Each kind of oil can be treated separately; so for each kind `i`, the `link` routine can call a subroutine `inner_link`. The link procedure is

```
link(continuous OldSt[KINDS],Purc[KINDS],Proc[KINDS],NewSt[KINDS])
{
      and(int k=0;k<KINDS;k++)
            innerlink(OldSt[k],Purc[k],Proc[k],NewSt[k]);
}
```

In the `innerlink` procedure, we need to state that the oil to be processed must be no more than the oil already in storage plus the oil purchased this month. Also, the available oil which is not processed each month is to be stored and made available for next month. These two conditions can be written as a pair of constraints, which gives us the procedure

```
innerlink(continuous OldSt,Purc,Proc,NewSt)
{
      Purc + OldSt >= Proc;
      OldSt + Purc - Proc == NewSt; // Subsumes previous constraint
}
```

Finally, we come to the procedure to calculate the revenue, which will be an exercise in double summations. Since the production process is conservative, we can compute the amount of oil produced as the sum of the individual vegetable oils processed. Given the entire `Plan` as a parameter, for each month `i` and each kind of oil `j`, the contribution to the profit will be the amount received for selling processed oil, minus the amount spent on buying raw oil and minus the cost of storing oil, which we can write as

```
      TONPRICE*Plan[i][PROC][j] -
            price[i][j]*Plan[i][PURC][j] -
                  STORCOST*Plan[i][STOR][j]
```

The net revenue will therefore be the double summation of the terms over `i` and `j`, giving us the procedure

```
objective_function(continuous Plan[][MODES][KINDS], Revenue,
                                             double price[][KINDS])
{
      Revenue ==
```

```
          sigma(int i=0;i<MONTHS;i++)
             sigma(int j=0;j<KINDS;j++)
                   ( TONPRICE*Plan[i][PROC][j] -
                         price[i][j]*Plan[i][PURC][j] -
                               STORCOST*Plan[i][STOR][j] );
}
```

There remains the output routine to develop and this is "left to the reader." The 2LP language supports a type `string` and elementary I/O. In order to carry out more sophisticated I/O operations for generating reports and the like, the thing to do is to use the functionality of C or other programming languages by calling external routines from the 2LP model.

The ability to integrate external routines and external storage is an important feature of a little language. As another example of this functionality, consider the case where the data of Tables 1 and 2 are already stored in external arrays. Then we would change the declarations to

```
extern double prices[][KINDS], double hardness[];
extern void data( double prices[][KINDS], hardness[] );
```

The 2LP interpreter is then passed the appropriate addresses by the surrounding C/C++ code. Further on, we will use this functionality in order to parallelize the treatment of an application.

4 Disjunction and Goal Programming

When one leaves the arena of classical linear programming, more flexibility is needed; for that we turn to the other logical connectives.

In mathematics, a disjunction can be known to be true without determining the truth or falsity of its disjuncts. Thus for example if x and y vary over the nonnegative real numbers and if we have shown that $xy \geq 100$, the statement

$$(x \geq 10) \vee (y \geq 10)$$

can be assumed to be true without first proving x to be ≥ 10 or else first proving y to be ≥ 10. This use of disjunction offers the mathematical problem solver the strategy of considering the case $x \geq 10$ and then coming back later to consider the case $y \geq 10$.

In classical programming languages this strategy for disjunction is not available; in order for the code to proceed through a disjunction, at least one of the disjuncts must first be determined to hold. Let us illustrate this point using the C/C++ notation for disjunction. If `a` and `b` are of type `double` or `int`, to evaluate a disjunction such as

```
(a >= 10 || b >= 10)        /* C fragment */
```

the interpreter first will perform the test `a <= 10`; if this succeeds, the second test is skipped and the program proceeds. If the first test fails, the second test will be performed. The system must determine explicitly that one of the disjuncts succeeds. Languages such as C, C++ and LISP employ a lazy strategy and will only evaluate as many disjuncts as are required to determine that one of them succeeds; once a disjunct succeeds, the program proceeds without leaving the option of coming back and considering an alternative disjunct. This interpretation of disjunction is called the *conditional or* or *c-or* [20]. Other languages such as Pascal follow a strict strategy and evaluate all the disjuncts whether or not one of them succeeds. But, in all cases, if the disjunction succeeds or fails, the program proceeds without leaving open the possibility of coming back and considering an alternative branch of the disjunction. In the next section, a programming approximation to mathematical disjunction will be considered, but first let us consider the extension of conditional disjunction to programming with constraints.

For conditional disjunction, the 2LP syntax is:

```
c_either <statement>
or <statement>
      ...
or <statement>
```

We will now apply this logical construct to a goal programming variation of the blending oils problem. Linear programming itself is oriented towards optimizing an objective function subject to a set of linear constraints. In practice, however, the decision maker often has several objectives to work on. If goals are prioritized, an incremental approach can be employed. Using 2LP this can be modeled in a straightforward way. Goals can be formulated and a program written to consider them in the order of their importance. The result is a sequence of successively refined linear programs.

The term *goal constraint* is used for a condition that is desirable but which is allowed to be violated to some extent. As noted above, with each feasible region constructed during the course of a 2LP program, there is associated a witness point which is a distinguished vertex on the region. The `X` coordinate of this point can be accessed by a call to `wp(X)`. This is a built-in function with signature

```
double wp(continuous)
```

Suppose that we are interested in a continuous variable `Quantity` and would like to make this variable reach or surpass some target goal. The variable `Quantity` labels one of the coordinate axes of the space in which the feasible region is defined. The following routine can be used to handle the goal constraint of making `Quantity` greater than or equal to goal:

```
try_to_stay_above_goal(continuous Quantity, double goal)
{
      c_either Quantity >= goal;       // Goal is met
      or {  // Do the best you can given that goal can't be reached
            max: Quantity;
            Quantity == wp(Quantity);
      }
}
```

A similar routine can be used to try to keep a continuous variable from exceeding a goal. Trying to make a continuous variable be as close as possible to a goal is also easily encapsulated:

```
try_to_stay_at_goal(continuous Quantity, double goal)
{
      if wp(Quantity) <= goal;         // Currently beneath goal
      then {
            Quantity <= goal;          // Stay beneath goal
            max: Quantity;             // Get as close as possible
            Quantity == wp(Quantity);  // Fix variable
      }
      else {  // Currently above goal
            Quantity >= goal;          // Stay above goal
            min: Quantity;             // Get as close as possible
            Quantity == wp(Quantity);  // Fix variable
      }
}
```

When an application involves goal constraints, developing a program for it is called *goal programming*. As an illustration of this, let us suppose that in our example, optimization of the objective function `Revenue` is too simple a formulation and the following goals are to be satisfied if possible:

1) Achieve a 6 month revenue of at least £95000
2) Aim to use about 30 tons of sesame oil in the blend every month
3) Keep the 6 month total usage of corn oil under 100 tons
4) Given all this, still to maximize revenue.

Goal 2 is really six goals, one for each month; let us prioritize these subgoals simply by placing the goals in order from the first month to the sixth month. We change the main procedure to

```
2lp_main()
```

```
{
      setup();                          // As before
      meet_goals(Revenue,Plan);         // New
      output(Plan,Revenue);             // As before
}
```

and write the meet_goals routine as follows:

```
#define CORN 0
#define SESAME 3

meet_goals(continuous R,P[][MODES][KINDS])
{
continuous T;           // Local continuous variable
                        // of static storage class

      try_to_stay_above_goal(R,95000);
      and(int i=0;i<MONTHS;i++)
            try_to_stay_at_goal(P[i][PROC][SESAME],30);
      T == sigma(int i=0;i<MONTHS;i++) P[i][PROC][CORN];
      try_to_stay_below_goal(T,100);
      try_to_stay_above_goal(R,1e6);
}
```

Note that we have formulated the fourth goal of maximizing revenue as the goal of making revenue as close to one million pounds as possible since optimization itself a form of trying to achieve a goal.

In the above example, we were able to prioritize the goals in a strict order. When two or more goals have equal priority, things are more complicated; this case can be handled using deviation variables. By way of example, if we do not want to prioritize the subgoals of Goal 2 in the example above, we would introduce new variables

```
continuous Under[MONTHS],Over[MONTHS];
```

and replace the code

```
      and(int i=0;i<MONTHS;i++)
            try_to_stay_at_goal(P[i][USED][SESAME],30);
```

with

```
and(int i=0;i<MONTHS;i++)
      P[i][USED][SESAME] + Under[i] - Over[i] == 30;
min: sigma(int i=0;i<MONTHS;i++) (Under[i] + Over[i]);
and(int i=0;i<MONTHS;i++)
      P[i][USED][SESAME] == wp(P[i][USED][SESAME]);
```

In effect, this forces the L_1 norm of the "error vector"

```
(Under[0]+Over[0],....,Under[MONTHS-1]+Over[MONTHS-1])
```

to stay at its minimum.

Taking a different tack, we could carry out a progressive minimization of the L_∞ size of the error vector. Let us describe how to do this. Let `Z` be an additional continuous variable. Add the constraints

```
and(int i=0;i<MONTHS;i++)
      Z >= Under[i]+ Over[i];
```

Then call

```
min: Z;
Z == wp(Z);
```

At this point, the feasible region is included in the union of the family of hyperplanes `Under[i] + Over[i] == wp(Z)`. If a polyhedron is covered by a finite set of hyperplanes, it must be completely contained in one of them, e.g. [28]. Therefore for at least one value of `i`, the minimum of `Under[i] + Over[i]` is fixed at `wp(Z)`. Now the process can be iterated on a smaller set of indices. Writing things out in detail, the `meet_goals` procedure becomes

```
new_meet_goals(continuous R,P[][MODES][KINDS])
{
continuous T,Under[MONTHS],Over[MONTHS];

      try_to_stay_above_goal(R,95000);
      and(int i=0;i<MONTHS;i++)
            P[i][PROC][SESAME] + Under[i] - Over[i] == 30;
      squeeze_infinity_norm(Under,Over);
      T == sigma(int i=0;i<MONTHS;i++) P[i][PROC][CORN];
      try_to_stay_below_goal(T,60);
      try_to_stay_above_goal(R,1e6);
}
```

The `squeeze_infinity_norm` routine unfolds as

```
#define OFF 0
#define ON 1

squeeze_infinity_norm(continuous Under[],Over[])
{
int count;
continuous Z[MONTHS];
int flags[MONTHS];

      count = 0;
      and(int i=0;i<MONTHS;i++)
            flags[i] = OFF;

       and(int j=0;count!=MONTHS;j++)
            chebychev(Z[j],Under,Over,count,flags);
}
```

Minimizing the L_∞ norm is known as *Chebychev goal programming.*

```
#define EPSILON 1e-6

chebychev(continuous Z, Under[], Over[],int & count, flags[])
{
      and(int i=0;i<MONTHS;i++)
            if flags[i] == OFF;
            then Z >= Under[i] + Over[i];

      min: Z;
      Z == wp(Z);

      and(int i=0;i<MONTHS;i++)
            if flags[i] == OFF;
            then {
                  min: Under[i] + Over[i];
                  if fabs(Under[i]+ Over[i] - Z) < EPSILON;
                  then {  // Trapped
                        flags[i] = ON;
                        count = count + 1;
                  }
            }
}
```

For an application of iterated Chebychev goal programming to problems from economics, see [34]. For yet another alternative, we could first minimize the L_∞ size of the error vector and then minimize its L_1 size. Then a progressive minimization of the L_∞ norm could be carried out. The interesting thing is that in interleaving these strategies, the minimization of the L_1 size of the error vector can only be done once and cannot be iterated the way minimization of the L_∞ norm can.

There is also a loop construct for conditional disjunction, the `c_or` loop, which is analogous to the `and` loop. As an illustration of the use of this loop construct, let us check to see if the pair $2x + z + 6 = 3y + 1$, $3x + 2z + 3 = 4y + 1$ of linear equations has an integer valued solution in the interval $[1, 100]$. This kind of question arises, for example, in work on compilers for parallel computation where it can be important to know if the indices in nested loops can become equal, e.g. [10]. The following code segment generates candidates until a solution is found:

```
c_or(int i=1;i<=100;i++)
     c_or(int j=1;j<=100;j++)
          c_or(int k=1;k<=100;k++) {
              2*i + k + 6 == 3*j + 1;
              3*i + 2*k + 3 == 4*j + 1;
              printf("A solution is (%d,%d,%d)\n",i,j,k);
          }
```

The search proceeds and enters the body of the innermost loop 10406 times until the mutual solution `i=2,j=5,k=6` is found. Let us see what happens when continuous variables are used. Assuming `X`, `Y` and `Z` are otherwise unconstrained, the following fragment uses continuous variables to find the same solution.

```
2*X + Z + 6 == 3*Y + 1;
3*X + 2*Z + 3 == 4*Y + 1;
c_or(int i=1;i<=100;i++){
      X == i;
      c_or(int j=1;j<=100;j++){
            Y == j;
          c_or(int k=1;k<=100;k++) {
                Z == k;
                printf("A solution is (%d,%d,%d)\n",i,j,k);
          }
      }
 }
```

With the continuous variables, the constraint `X == i` fails for `i=1`, and so the inner loops are not entered until `i` reaches `2`; then the constraint `Y == j` fails for all `j < 5` and the innermost loop is not entered at all. Finally, the first two constraints are satisfiable at `i=2`, `j=5`, the body of the innermost loop is entered 7 times succeeding at `k=6`. The technique of establishing the constraints before trying to generate values for the variables is known as constrain-and-generate, see [30]. The previous technique of generating values and then testing if the constraints are satisfied is called *generate-and-test*, see [38].

5 Persistent Disjunction and Disjunctive Programming

As remarked previously, in ordinary mathematics, a branch of a disjunction can be explored and other branches held in abeyance so that they can be returned to later if necessary. A version of this kind of disjunction is supported in 2LP; this is done by means of the `either/or` construct, whose syntax is:

```
either <statement>
or <statement>
      ...
or <statement>
```

The `either/or` construct blends the mathematical and the programming versions of disjunction: to proceed past a disjunction one of the disjuncts must be committed to but the alternative of coming back and considering further disjuncts is maintained. This is called *persistent disjunction.*

Let us look at the following disjunction of constraints:

```
either X <= 3;
or X >= 9;
```

The declarative meaning of this is that, from this point in the program, `X` is restricted to vary over the union of the intervals $[0,3]$ and $[9,+\infty)$. To capture this meaning, the operational semantics of 2LP treats this disjunction as a statement with two alternative parts. When encountering the call to `either/or`, the system will put this call on its program stack and record the fact that this statement has a second alternative. We say that a *choice point* has been created. The program first tries the constraint `X <= 3`; if it is not feasible, the alternative `X <= 9` is tried straightaway. If `X <= 3` is feasible, the program will commit to it and proceed to the next statement. The existence of the choice point will enable the program to return later, undo the constraint `X <= 3`, and try the alternative. This is called *backtracking.*

In 2LP, conjunction can be expressed both by juxtaposition and by the `and` loop; conditional disjunction has the `c_either/or` construct and the `c_or` loop.

Analogously, persistent disjunction can also be expressed in two ways: by means of the `either/or` construct and by means of the `or` loop. The key features that provide the programming support for persistent disjunction in 2LP are choice points and the backtracking mechanism. When a choice point is returned to in a program, backtracking restores the feasible region and the loop control variables to their original state. Choice points are *robust*: when a choice point is created within a procedure, it stays open even after the procedure has been exited; if backtracking takes place, the procedure is re-entered and processing begins again with the next untried branch of the choice point.

Persistent disjunction can serve as an approximation to the working problem solver's periodic need to start all over again. The connection between the way persistent disjunction restores the state of the continuous variables and the loop control variables and the way one solves problems can be understood informally: Consider the mathematician who reaches an environment in the argument where a certain number of lemmas have been established, where bounds on variables have been determined, etc. At that point, there might be several possibilities to explore. One way is chosen. The ensuing argument tightens bounds and further narrows things until a conclusion is reached or the line of reasoning fails. In case of failure, the reasoning jumps back to the environment that was in force when the failed path was first taken discarding all intermediate results. This can be emulated in a robust way by persistent disjunction: Upon backtracking, the feasible region and the loop control variables are restored and the program starts again where it left off.

Persistent disjunction can also serve as an approximation to classical disjunction. For discussion, see [30]. For research on connections between classical logic and procedural programming mechanisms, to start see [21] and [26].

The 2LP interpreter, which supports both linear constraints and persistent disjunction, is based on two fundamental technologies. The type `continuous` is maintained by means of an incremental linear programming solver based on the revised simplex method. The backtracking and choice point machinery is based on the Warren Abstract Machine architecture, see [36], [1]. This architecture is commonly called the WAM. The 2LP interpreter employs an appropriately engineered version of the WAM, which is adapted to support the types `int`, `double` and `continuous`; it is a virtual machine which is emulated in C. A technical point that might be of interest: in the implementation of 2LP of [30], the linear programming module of 2LP does not stack simplex bases for backtracking; instead, a trail of changes to the polytope is kept and the changes are undone as needed. A 2LP program is compiled into code for this virtual machine. The theoretical analysis that underlies the 2LP implementation can be found in [13].

Languages which support logic programming constructs and constraints are called *constraint logic programming* languages or *CLP* languages, [23]. The first to support constraints on continuous variables were Prolog III, [12] and CLP(R), [25]. For work on CLP languages which support constraints over discrete domains, see [14] and [35]. For a survey, see [24].

We now introduce an important class of constrain-and-generate applications, known as *disjunctive programs*, [8], [9], [19]. The special mathematical form of these applications will allow for greater flexibility in carrying out search strategies.

A union of polyhedral regions is called a *disjunctive set*. The problem of determining whether the intersection of a family of disjunctive sets is non-empty is called a *disjunctive program*. In mathematical notation the problem is to determine if a set of the following form is non-empty

$$\bigcap_{i<M} \bigcup_{j<N} F_{ij}$$

Typically in a disjunctive program application, each $i < M$ will index a logical requirement L_i and each disjuctive set $\cup F_{ij}$ will arise as a set of alternative ways of meeting that requirement. The question in a disjunctive program is whether there are solutions which satisfy all these requirements simultaneously. When an objective function is present, the question becomes that of both determining whether a solution exists and if so, finding the best possible solution. *Disjunctive constraint* is a term often used to denote a logical requirement which is expressed as a disjunctive set; to avoid confusion with the usual sense of constraint, we will use the expression *disjunctive requirement* or *logical requirement.*

Note that a disjunctive program has the structure of Conjunctive Normal Form and one can see its relationship with Satisfiability, the quintessential NP-Complete problem. In terms of constraints, a feasible region is defined by a conjunction of constraints. A union of feasible regions is given by a persistent disjunction of these conjunctions. Then the intersection of these unions is given by a conjunction. More concretely,

$$\bigcap_{i<M} \bigcup_{j<N} F_{ij}$$

is modeled by

```
and(int i=0;i<M;i++)            // Intersection
      or(int j=0;j<N;j++)       // Union
          cc(i,j);              // Fij
```

where `cc(i,j)` generates the conjunction of constraints that defines the feasible region F_{ij}. Thus, in practice, disjunctive programs take the form of a constrain-and-generate application, with a setup routine restricting the program to an initial feasible region:

```
setup();          // Constraints for initial feasible region
and(int i=0;i<M;i++)       // Intersection
      or(int j=0;j<N;j++)  // Union
            cc(i,j);       // Fij
```

The initial setup constraints provide a linear relaxation that will be tightened as the code progresses through the combined `and/or` loop.

With a disjunctive program, the computational task is to determine whether a solution exists and if so, to produce a feasible region that provides a solution; more precisely, finding a solution means determining a sequence $j_0, ..., j_{M-1}$ such that the polyhedron

$$F \cap F_{0j_0} \cap \ldots \cap F_{M-1 j_{M-1}}$$

is non-empty. A vertex on this polyhedron will then provide a numerical solution to the problem. In programming terms this means that the sequence of calls, `cc(0,j`$_0$`)`,...,`cc(M-1,j`$_{M-1}$`)` will succeed and the witness point will furnish a concrete solution. Let us note that, although there are only $M \times N$ feasible regions F_{ij} in the formulation of the disjunctive program, the number of sequences $j_0, ..., j_{M-1}$ that are candidates to generate a solution is N^M. It is here that we encounter combinatorial explosion.

Disjunctive programs have a nice mathematical property which is very important from a computational point of view. Since set intersection is a commutative operation, permuting the indices $i < M$ does not change the solution set. Similarly, set union is commutative and, for given i, nothing is changed if the indices $j < N$ are permuted. Thus, for example, the above program can also be written

```
setup();      // Constraints for initial feasible region
and(int i=M-1;i>=0;i--)         // Permute the i
      or(int j=N-1;j>=0;j--)    // Permute the j
            cc(i,j);
```

Mathematically, permuting indices doesn't change the underlying set of solutions. In terms of programming, permuting the indices can mean that a different solution will be found if there is one. It can also mean a difference in the time required to find a solution or in the time to determine that there is no solution.

To illustrate disjunctive programming we return to the blending problem and its second version in [37]. Management is not happy with the plan determined by the linear program nor with the goal program, basically because several key elements of the process were not taken into account. It turns out that there are some logical requirements on each month's production, to wit:

1) No more than 3 oils can be used in each month's production run.

2) If an oil is used in a monthly production run, then at least 20 tons of it must be used.

3) If either sesame or sunflower oil is used in the blend, then corn oil must also be used.

In [37] these conditions form the second version of the problem, one which is described as "difficult to solve" from a computational point of view.

This is still a 6 month planning problem but now each month also has a set of logical requirements. For each month and for each oil it must be determined whether that oil is to be included in the product blend and these determinations must satisfy requirements 1), 2) and 3) for each month. The first question is whether these conditions can be met. That done, we can address the issue of finding an optimal solution.

The logical requirements are imposed on each month's production and so we can take things a month at a time. For each oil it must be decided whether or not to include that oil in the blend. The corn oil plays a special role, since if it is not included then the last two oils, sesame and sunflower, can not be included. To handle these different oils specifically, let us introduce symbolic constants for them and names for "in" and "out" and for "20.0" and "0.0":

```
#define CORN 0
#define PEANUT 1
#define SAFFLOWER 2
#define SESAME 3
#define SUNFLOWER 4

#define IN 0
#define OUT 1
#define THRESHOLD 20.0
#define ZERO 0.0
```

The main procedure will be

```
2lp_main()
{
      setup();
      simple_search();
      max: Revenue;    // There is still a linear program to solve
}
```

Let us turn to the task of coding the search routine. A straightforward way to organize the search is to go through the months and the oils deciding whether or not to include the oil in the blend. Thus the routine can take the following form:

```
simple_search()
{
      and( int i=0;i<MONTHS; i++)
            and(int k=0;k<KINDS;k++)
                  either use_oil(k,Plan[i][PROC]);
                  or dont_use_oil(k,Plan[i][PROC]);
}
```

Since it will be necessary to monitor the requirement that at most 3 oils figure in the blend each month, it will be useful to have a routine to check that at least `i` of the next `j` oils among `Oils[m]`,...,`Oils[KINDS-1]` have zero witness point values:

```
zero(int i,j,m, continuous Oils[])
{
      if i == 0;          // Enough zeroes accounted for ?
      then return;
      j >= i;             // Check room left to find enough zeroes
      if wp(Oils[m]) < EPSILON;    // Another zero found ?
      then zero(i-1,j-1,m+1,Oils); // A recursion
      else zero(i,j-1,m+1,Oils);   // Ditto
}
```

The following procedure is passed a vector `Oils` and an integer `k` $<$ `KINDS` such that `Oils[0]`,...,`Oils[k-1]` have been dealt with:

```
use_oil(int k,continuous Oils[])
{
      Oils[k] >= THRESHOLD;        // Use Oil[k]
      if {
            k == 2;                // SAFFLOWER;
            not zero(1,2,0,Oils);  // All previous oils non-zero ?
      }
      then {                       // Last two oils cannot be used
            Oils[k+1]==0;
            Oils[k+2]==0;
      }
      if {
            k == 3;                // SESAME
            not zero(2,3,0,Oils);  // Two previous oils non-zero ?
      }
      then Oils[k+1]==0;           // Last oil cannot be used
```

```
}

dont_use_oil(int k, continuous Oils[])
{
      Oils[k] == 0;
      if k == CORN;
      then {     // Excluding CORN means excluding the last two
            Oils[SESAME] == 0;
            Oils[SUNFLOWER] == 0;
      }
}
```

The model requires 91 continuous variables and 60 constraints. If coded in classical MIP fashion, the model requires 30 additional 0-1 variables and an additional 78 (multivariable) constraints. In this example, the added machinery will not "tighten" the model.

The code determines that a solution indeed exists but no effort has been made to find a quality solution. Let us look into the question of finding the optimal solution. The backtracking support for persistent disjunction in 2LP makes it straightforward to turn `simple_search` into a depth-first branch-and-bound search. In fact, 2LP has built-in constructs **`find_max`** and **`find_min`** for carrying this out automatically. In short, the optimal solution to the disjunctive program can be found with the code

```
2lp_main()
{
      setup();
      find_max: Revenue;
      subject_to simple_search();
      output(Plan,Revenue);
}
```

The optimal solution is found to be £100278. The simple search will carry out a bovine enumeration of the search space and the only pruning mechanism will be the bound on the objective function applied to the linear relaxation. However, the optimal solution is found and verified quickly enough on a PC. In earlier editions of [37], the optimal solution is reported as found but not verified, even with state-of-the-art technology due to time limitations.

If the planning horizon is lengthened to 12 months or broken down bi-monthly over 6 months, the only changes that must be made in the code are the definition of the symbolic constant `MONTHS` and the initialization of the data array `prices` which now includes the data for the additional months or periods. However, with this bigger problem the simple search strategy can prove inadequate. In the next sections, we discuss ways of making the search process more intelligent.

To close this section, let us note that in traditional programming languages such as C and C++ the functionality of persistent disjunction for discrete applications is captured by conditional disjunction and recursion; in LISP and Scheme, it can be done with continuations; in Prolog it is done by means of multiple definitions of a procedure.

6 Negation and Hill Climbing

We come to the fourth of the basic logical connectives, negation. For working with constraints, the semantics of negation is based on negation-as-failure, a concept used in database work and in logic programming. This concept extends the semantics of negation encountered in C, C++ and other programming languages to cover constraints.

In classical symbolic logic, the fundamental property of negation is involution

$$\neg\neg p \text{ is equivalent to } p$$

However, this property of negation has stirred controversy among mathematicians and, for example, it is not valid in intuitionistic logic and other branches of constructive mathematics; see [16].

In classical programming languages, negation simply inverts the truth value returned and therefore the involutive property of negation is maintained. Something else can happen to the numerical values returned; for example, in C the expression `!!7` evaluates to `1`.

Formally, negation in 2LP has the syntax

```
not <statement>
```

and the semantics of `not <statement>` is that it succeeds if `<statement>` fails and conversely, the negation fails if `<statement>` succeeds.

Operationally, `not <statement>` is equivalent to

```
if  <statement>
then 0 == 1;  // Always fails
```

In other words, when a call to `not <statement>` is made, `<statement>` itself is executed and then its truth value is reversed and the feasible region is restored to its previous state. Variables of type `int` and `double` are not restored and retain the values set in the call to `<statement>`. The critical thing is that no matter if not `<statement>` succeeds or fails, the feasible region is restored to the state it had before the call was made.

Since negation does not change the feasible region, it can be used to test the consistency of sets of constraints and of procedures. For example, the code segment

```
if not X <= Y;
then printf("The constraint would fail\n");
else printf("The constraint would succeed\n");
```

will determine if the constraint `X <= Y` could be successfully added to the feasible region; the code segment does this without changing the current feasible region. This is quite different from the code

```
if X <= Y;
then printf("The constraint has succeeded\n");
else printf("The constraint has failed\n");
```

which changes the feasible region by adding the constraint `X <= Y` if it is possible to do so.

For a procedure `p()` the code

```
if not p();
then printf("A call to p() can not be successful\n");
else printf("A call to p() can be successful\n");
```

determines if `p()` can be called successfully; again this is done without changing the current feasible region.

The term *heuristic* is used in different ways in the literature. On the one hand, the term denotes a search strategy which yields an acceptable solution to a difficult application, though not necessarily the best possible solution. On the other hand, a heuristic is a guide used to determine which branch of a search tree to explore next. In this section we develop strategies where the next move chosen is the one that some heuristic criterion deems the most attractive. The heuristic measure used will be computed by an evaluation function. There are two kinds of evaluation based search strategies: (1) those where one alternative is chosen and the rest are discarded and (2) those where some or all of the unchosen alternatives are kept in abeyance. The former are called *hill climbing* strategies. If all alternatives are kept for further consideration and the most promising of all considered next, the search is called *best first*.

By way of illustration, suppose that `future1()` and `future2()` are routines that develop alternative successor nodes for a given node in a generate-and-test model. Suppose that `Z` is an objective function and that we are seeking a solution which makes `Z` large. Then at any point in the search, the maximum value of `Z` over the current linear relaxation provides us with a natural evaluation function. So confronted with two possible choices for the future, the following code enables us to look into each of these possible worlds and to compare them.

```
look_into_the_future(double & cache1,& cache2)
{
      if not {
            future1();              // Go into the first future
            max: Z;                 // Optimize
            cache1 = wp(Z);         // Record evaluation
      }
      then cache1 = -INFINITY;      // In case future1() fails

      if not {
            future2();              // Go into second future
            max: Z;                 // Optimize
            cache2 = wp(Z);         // Record evaluation
      }
      then cache2 = -INFINITY;      // In case future2() fails
}
```

When this code returns, an evaluation of the two futures will be contained in the storeback variables `cache1` and `cache2`, and the current feasible region will not have changed. Now one of the futures can be selected based on these evaluations for a hill climbing search or they can be prioritized for a complete search.

We now return to our model. The program will use a hill climbing strategy to evaluate the two alternatives of including an oil in the blend or not. The alternative with the better evaluation will be taken.

Recall from above that the amounts of oil to be processed in month `i` are represented by the vector `Plan[i][PROC]`. Working top-down, we can formulate the heuristic search routine as

```
hillclimb()
{
      and(int i=0;i<MONTHS;i++)
            and(int k=0;k<KINDS;k++)
                  steeper_gradient(i,k,Plan[i][PROC]);
}
```

where `steeper_gradient` will look one level (or more) into the future to evaluate the alternatives of using an oil in the blend or not.

The next routine is a standard look into the future to choose between two alternatives.

```
steeper_gradient(int i,k,continuous Oils[])
{
double u, uu;

      if not {
            use_oil(k,Oils);
            max: Revenue;
            u = wp(Revenue);
      }
      then u = 0;

      if not {
            dont_use_oil(k,Oils);
            max: Revenue;
            uu = wp(Revenue);
      }
      then uu = 0;

      if u >= uu;                   // Bold code
      then use_oil(k,Oils);
      else dont_use_oil(k,Oils);
}
```

To hedge one's bets and do a complete search, the bold code can be replaced with the following:

```
      if u >= uu;
      then          either  use_oil(k,Oils);
                    or dont_use_oil(k,Oils);
      else          either dont_use_oil(k,Oils);
                    or use_oil(k,Oils);
```

The main routine will be as in the model above with a conditional call to the `hillclimb` routine inserted as follows:

```
2lp_main()
{
      setup();
      if hillclimb();
```

```
      then {
          max: Revenue; // There is still a linear program to solve
          output(Revenue,Plan);
      }
      else printf("The hill climb did not succeed\n";
}
```

For 6 month, 12 month and larger versions of the problem, this kind of heuristic performs very well.

7 Disjunctive Programming and The Injury Method

The next step to take is to use the result obtained by the hill climbing search as a bluff for a branch-and-bound search to find and verify the best possible solution for large problems. For that, we can reorganize the main procedure to follow the hill climb with a branch-and-bound search:

```
double cache;          // New global variable for storing bounds

2lp_main()
{
      setup();
      heuristic();
      find_max: Revenue;    // Built-in construct for backtracking
      subject_to            // to find better and better solutions
          simple_search();  // From Section 5

      output(Plan,Revenue);
}

heuristic()
{
      if not {
            hillclimb();
            max: Revenue;
            cache = wp(Revenue);      // Record result
      }
      then {
            printf("The hill climb did not succeed\n");
            cache = 0.0;
      }
```

```
    else printf("Heuristic found objective of %f\n",cache);

    Revenue >= cache;                // A guaranteed bluff
}
```

This code again will find the true optimum solution, albeit very slowly on a large problem. As the reader can imagine, running the branch-and-bound search without the cutoff value found by the heuristic search can take even longer. However, branch-and-bound search is a delicate matter and great improvements in performance can be had by matching the semantics of the application with an astute search strategy. So let us develop tools for more sophisticated searches.

Now we present an important technique, the *injury method*, for organizing the branching strategy in the branch-and-bound search to solve a disjunctive program to optimality. This method takes advantage of the mathematical structure of disjunctive programs to reshape the search tree dynamically. The basic idea is found in [27] and [15]. In MIP programming, it reduces to the strategy for choosing the branching variable. The terminology we use here is adapted from the branch of Mathematical Logic known as Recursion Theory.

If we want to improve the branching strategy, the best thing would be to eliminate branching altogether, and failing that to keep it to a minimum. A disjunctive program typically consists of setup constraints followed by a list of disjunctive logical requirements $L_0, ..., L_{M-1}$, where each of these requirements is expressed as a disjunction of conjunctions of constraints. The requirement L_i corresponds to a disjunctive set $\cup F_{ij}$. Branching is usually needed to force these requirements to be satisfied by selecting one of the F_{ij} for each i. The program is typically centered around an `and/or` loop

```
    and(int i=0;i<M;i++)            // For each i
          or(int j=0;j<N;j++)       // find j to satisfy Li
                cc(i,j);            // by means of Fij
```

where `cc(i,j)` is a procedure for generating a conjunction of constraints which meets the requirement L_i, namely the one which generates the feasible region F_{ij}. The idea of the injury method is to use the current witness point to determine which requirement to address next, rather than following the given order $0, ..., M-1$. This amounts to a heuristic reordering of the requirements. The goal is to reach a node where the solution to the linear relaxation to the problem also satisfies the logical requirements. When this happens, no additional branching is needed.

If the witness point does not satisfy L_i we say that this requirement is *injured.* The following template outlines the opportunistic strategy, known as the *injury method* for the optimization form of a disjunctive program:

```
and(;;) {
      <optimize the objective function>
      if <there are no injured requirements>
      then break;
      <determine an injured requirement>
      or(int j=0;j<N;j++)
            cc(injured,j);
}
```

This strategy is opportunistic in that the code will exit once a witness point is reached where no requirements are injured; in this way unnecessary branching can hopefully be avoided. To use the injury method in an optimization problem, we just have to make sure that the check for injuries is made when the witness point is one which is optimal for the linear relaxation before checking for injuries. Then one can put the search code in a `find_min` or `find_max`. For an analysis of the soundness and completeness of this algorithm, see [30].

If the model is a MIP model, injuries reduce to variables' being non-integral and the injury method is that much easier to code. Here we have a more complex example which will serve to illustrate the method in greater generality. We will also make use of some simple penalty computations [33] to tighten the model and to illustrate how they can be readily applied to disjunctive programs.

From the point of view of coding, the main routine only needs to change the call to the search routine.

```
2lp_main()
{
      setup();
      heuristic();
      find_max: Revenue;
      subject_to
         smart_search();
}
```

The basic unit of this model is the month of production. Therefore we will organize the injury method around the months. A month is injured if one of the requirements 1), 2), 3) is violated. A simple way to structure an injury is by means of the oils themselves, starting with corn oil. The following routine is passed a months production `Plan[i][PROC]` and an integer `j`; the integer `j` will keep track of which oils have already been repaired.

```
#define EPSILON .000001
```

```
a_ok(int j, continuous Oils[])      // Succeeds if no injuries
{
      and(int i=j;i<KINDS;i++)
            if wp(Oils[i]) > EPSILON;               // Condition 2
            then wp(Oils[i]) >= THRESHOLD - EPSILON;
      if {
            j==0;                                   // Condition 3
            wp(Oils[j]) < EPSILON;
      }
      then and(int k=CPS;k<KINDS;k++) wp(Oils[k]) < EPSILON;
      else zero(2,KINDS,0,Oils);                    // Condition 1
}
```

To keep tabs on which injuries have been repaired we use a technique known as "marking and trailing." An array `depth` will mark how far into each month the repairs have been made; the stack `trail` will record that changes have made to the array `depth`. When backtracking takes place, the `trail` stack will be popped and `depth` will be restored to its proper state.

As the search progresses, penalty computations will be made and variables that can be forced to stay at 0.0 or above the threshold of 20.0 will be detected and fixed accordingly.

```
#define GAP 100.0         // To move solutions along
smart_search()
{
int eye, current_month;
int depth[MONTHS], trail[MONTHS*KINDS], top;

      top = -1;                           // Empty trail stack
      and(int i=0;i<MONTHS;i++)
            depth[i] = 0;                 // Lowest depths
      current_month = MONTHS-1;           // Most profitable month
      absgap(GAP);        // Avoid virtually duplicate solutions
      and(int m=0;;m++) { // m records depth reached in search tree
            penalty_computation(depth,GAP);   // To tighten model
            if find_injury(eye,current_month,depth);
            then
              tend_to_month(m,eye,Plan[eye][PROC],depth,trail,top);
            else break;                       // Solution found
      }
      printf("Solution at %f\n",Revenue);   // Print to screen
      cache = wp(Revenue);                  // New bound
```

```
}
```

To distribute the search for injuries more evenly among the months, we follow a circular loop among the months `0,...,MONTHS-1` in the hunt for an injured month. The integer variable `current_month` will keep track of the month among `0,...,MONTHS-1` to start with. In essence this needs an "until" loop, one with exit condition `current_month==(i+1)%MONTHS` but which must be executed at least once. We get this effect by running the test `current_month != (i+1)MONTHS` inside an `and` loop.

```
find_injury(int & eye, & current_month, int depth[])
{
      and(int i=current_month; ;i=(i+1)%MONTHS) {
            if not a_ok(depth[i],Plan[i][PROC]);
            then {
                  eye = i;
                  current_month=(i+1)%MONTHS; // New month
                  return;
            }
            current_month  != (i+1)%MONTHS;   // Circled round?
      }
}
```

If month `i` is injured, we know that the injury can only begin at the oil `depth[i]`:

```
tend_to_month(int m,i,continuous Oils[],int depth[],trail[], & top)
{
int j;
      j = depth[i];         // Push stacks
      depth[i] = depth[i]+1;
      top = top+1;          // Same as top = m
      trail[top] = i;
      if wp(Oils[j]) >= THRESHOLD;
      then {
            either use_oil(j,Oils);
            or {
                  and(int t=top;t>m;t--)        // Clean up stacks
                        depth[trail[t]] = depth[trail[t]]-1;
                        top = m;
                        dont_use_oil(j,Oils);
            }
```

```
    }
    else {
          either dont_use_oil(j,Oils);
          or {
                and(int t=top;t>m;t--)         // Clean up stacks
                      depth[trail[t]] = depth[trail[t]]-1;
                top = m;
                use_oil(j,Oils);
          }
    }
}
```

Finally we get to the penalty computations. First we need an auxiliary routine to detect if a variable is not in position to be assessed a penalty.

```
ineligible(continuous Oil)
{     // lb and ub are built-in functions that
      // return the bounds on a variable
      c_either lb(Oil) >= THRESHOLD;     // Oil already used ?
      or ub(Oil) == ZERO;                // Oil already eliminated ?
}
```

Then we use the reduced cost function `rc` to compute the penalties. If a variable is at its lower bound of 0.0, then the reduced cost multiplied by 20.0, the threshold, will give a conservative estimate of how much the objective function will deteriorate if the variable is moved above 20.0. If the linear optimum less this estimate falls below the current bound which is `cache + GAP`, the variable can be fixed at 0.0. A similar remark applies if the variable is currently at its upper bound. The function call `rc(X)` returns the reduced cost of `X` if `X` is at its upper bound and the reduced cost multiplied by -1 if `X` is at its lower bound. If `X` is between bounds, it is "in the basis" and has reduced cost 0.0, which is the value returned.

```
penalty_computation(int depth[], double gap)
{
double cwp,src,crev;
int eye[MONTHS*KINDS];
int jay[MONTHS*KINDS];
int dir[MONTHS*KINDS];
int apex;
      apex = 0;
      max: Revenue;        // Linear optimization for reduced costs
```

```
      crev = wp(Revenue); // Record linear optimum
      and(int i=0;i<MONTHS;i++)
            and(int j=depth[i];j<KINDS;j++) {
                  if ineligible(Plan[i][PROC][j]);
                  then continue;     // Move on if ineligible
                  cwp = wp(Plan[i][PROC][j]);  // Current value
                  src =  rc(Plan[i][PROC][j]); // Reduced cost
                  if crev + THRESHOLD*src < cache + gap;  // Bingo
                  then {                        // Record variable
                        eye[apex] = i;
                        jay[apex] = j;
                        dir[apex] = 0;    // To fix at lower bound
                        apex = apex + 1;
                  }
                  else {
                        if crev - src*cwp < cache + gap;   // Bingo
                        then {                // Record variable
                              eye[apex] = i;
                              jay[apex] = j;
                              dir[apex] = 1; // Enforce threshold
                              apex = apex + 1;
                        }
                  }
            }
      and(int k=0;k<apex;k++)           // Use results
            if dir[k] == 0;
                  then Plan[eye[k]][PROC][jay[k]] == ZERO;
                  else Plan[eye[k]][PROC][jay[k]] >= THRESHOLD;

      max: Revenue;                    // For injury method
}
```

Penalty computations are among the earliest techniques developed for branch-and-bound search. The version employed in the above model is relatively simple and has small computational overhead. More elaborate penalty computations can be made by extracting more information from the simplex method, see [33]; these more elaborate methods mean greater computational overhead but can apply to variables that are currently between their lower and upper bounds. Other kinds of pseudo-cost computations can also be brought into play.

The net effect of all this is that the optimal solution can be found very quickly on the original 6 month problem and reasonably quickly on larger versions of the problem.

8 A Randomized Shuffle

Randomization is playing a more and more prominent role in problem solving. Here we present an algorithm that uses branch-and-bound optimization within a randomized local search.

Suppose we extend the scope of our blending problem to 24 months or periods. We now have price data for additional periods. However, the branch-and-bound search to find the optimal solution can be overwhelmed by the size of the problem. Let us explore a way of handling this.

Let us suppose that through a hill climb or other means we have an initial solution which is stored in the array `plan[MONTHS][MODES][KINDS]` of type `double` and suppose that the value of the objective function is again stored in `cache`. The idea will be to commit to a random subsolution of `plan` and to find an optimal extension of the subsolution. This process can be iterated any number of times. The quality of the solutions found will be non-decreasing.

The following routine is passed an integer array of length `MONTHS` and randomly fills the array with 0s and 1s. The idea is that a month with corresponding entry 1 will be fixed at the values recorded in the array `plan`, while the remaining months will be left free.

```
subsolution(int q[])
{
int count;

      // The random function returns a value in (0,1)
      // The nint function returns the nearest integer
      and(int i=0;i<MONTHS;i++)
            q[i] = nint(random());
      count = 0;
      and(int i=0;i<MONTHS;i++)
            if q[i] == 1; then count = count +1;

      if count < MONTHS/3;           // Too few 1's ?
      then subsolution(q);           // Try again
}
```

Committing to the subsolution is straightforward:

```
commit_to(int q[])
{
      and(int i=0;i<MONTHS;i++)
           if q[i] == 1;
```

```
           then
                  and(int j=0;j<KINDS;j++)
                        and(int k=0;k<MODES;k++)
                              Plan[i][k][[j] == plan[i][k][j];
}
```

The local search code is

```
local_improvement()
{
int q[MONTHS];

      subsolution(q);
      commit_to(q);
      Revenue >= cache;            // A guaranteed bluff
      if find_max: Revenue;        // Try to solve tail of
                                   // problem to optimality
      subject_to smart_search();   // respecting GAP requirement
      then record_plan();          // Enter in plan and cache
}
```

The number of iterations must be defined. It is also convenient to define a macro for double negation-as-failure; `not not` has the effect of running code and then restoring the feasible region to its original state, which is what is needed to relaunch the local search.

```
#define ITERATIONS 33
#define do_record_undo not not
```

In `2lp_main`, after running a heuristic to find an initial solution, we seed the random number generator and look for better and better solutions:

```
2lp_main()
{
      setup();
      run_heuristic_and_record_solution();   // To be filled in
      seed(2393);
      and(int n=0;n<ITERATIONS;n++)
            do_record_undo
                 local_improvement();
}
```

If needed, more diversification in the search can be obtained by running different versions of the heuristic to find different starting positions for the local search.

This method is an example of a *shuffle*. For an application to job shop scheduling, see [2].

9 Distributed Programming Libraries

In the context of parallel programming, 2LP has two key functionalities: on the one hand, persistent disjunction and the logic programming control structure give the programmer the ability to subdivide the application into parts of varying granularity; on the other, the fact that functions from a distributed programming library can be called and data shared from a running 2LP model means that a 2LP program can be parallelized without any change to the 2LP system itself. This merger can carried out by linking to a message passing library for distributed processing such as PVM [18], MPI [22] or DP [3].

Details of the merger of DP and 2LP are reported in [4] and [5]. Work with PVM is given in [7]. Because 2LP has a logic-based control, the messages that need to be passed between processes, including those for assigning new tasks, are very compact. This is critically important in a message-passing parallel environment. What we mean here is that simplex bases and other node data structures do not need to be passed between processes; instead, instructions are sent which the worker process uses to build the node locally.

Parallelization of a 2LP application can take place at several levels. One kind of situation arises in applications which are disjunctive programs which split into a tree whose leaf nodes are linear or integer programs; this occurs, for example, in financial models when regulations permit different structurings of a model and the client can determine which is most beneficial. At yet another level, in applications where logical or integrality requirements lead to large search spaces, parallelism can be used to divide the search among cooperating processes. In this situation, the search can at times be pruned dramatically as processes communicate results obtained.

At the simplest level, the application will consist of a collection of virtually independent linear, integer or disjunctive programs. This is the situation we will look at briefly. Let us consider a stochastic version of the blending problem. Suppose we are given different randomized data for the prices over the next period of months. If one makes independent runs of the same model on different data sets, the results can be assembled at the end for ex post analysis. Here the parallel control need only launch the separate processes and assemble the results at completion.

The task is to analyze how much revenue can be expected to be achieved given various pricing scenarios for the coming months. These pricing scenarios can be obtained from the surrounding C/C++ code. By distributing effort among worker processes, large numbers of these scenarios can be run with virtually linear speedup. In the `2lp_main` routine, the process enters an indefinite `or` loop obtaining data and reporting results.

```
... definitions ...

extern double price[][KINDS];                              // New
extern void getPrices(p[][KINDS]);                         // New
extern void sendResult(double & c, p[]PROC][KINDS]);       // New

double hardness[KINDS];
continuous Revenue, Plan[MONTHS][MODES][KINDS];
double cache;
double plan[MONTHS][MODES][KINDS];
#define fail 0==1

2lp_main()                              // Worker's code
{
      // Underlying linear model
      set_up_less_objective(Plan);

      or(;;)
      {
            getPrices(price); // New data or stop order from master
            objective(Plan,Revenue,price);
            find_max: Revenue;
            subject_to smart_search();
            record_plan(cache,plan);
            sendResult(cache,plan);     // Send to master process

            fail; // Undo solution and go to top of loop
      }
}
```

For more detail and other applications, see [5].

10 Concluding remarks

The 2LP system currently runs on UNIX workstations and Windows PCs. These systems, models, and related papers can be obtained from the home page of the Logic Based Systems Lab, http:// www.brooklyn.cuny.edu/lbslab.

The authors would like to acknowledge the support of ONR grant N0014-94-1-0035.

References

[1] H. Ait-Kaci, *Warren's Abstract Machine: A Tutorial Reconstruction*, MIT Press, 1991.

[2] D. Applegate and W. Cook, A Computational Study of the Job-Shop Scheduling Problem, ORSA Journal on Computing 3, No. 2 (1991) pp 149-156.

[3] D.M. Arnow, DP - a library for building reliable, portable distributed programming systems, in *Proceedings of the USENIX Winter '95 Technical Conference*, 1995.

[4] D. M. Arnow, K. McAloon, and C. Tretkoff, Disjunctive programming and distributed programming, in *Proceedings of the Sixth IASTED-ISMM Conference on Parallel and Distributed Computing and Systems*, 1994.

[5] D. M. Arnow, K. McAloon, and C. Tretkoff, Parallel integer goal programming, in *Proceedings of the 23rd Annual ACM Conference*, 1995.

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

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

[8] E. Balas, Disjunctive programming and a hierarchy of relaxations for discrete optimization problems, SIAM J. Alg. Disc. Meth. 6 (1985) pp 466-486.

[9] N. Beaumont, An algorithm for disjunctive programming, European Journal of Operational Research 48 (1990) pp 362-371.

[10] R. Bixby, K. Kennedy and U. Kremer, Automatic Data Layout Using 0-1 Integer Programming, in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 1994.

[11] A. Brooke, D. Kendrick and A. Meeraus, *GAMS: A User's Guide*, The Scientific Press, 1988.

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

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

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

[15] R.J. Dakin, A tree-search algorithm for mixed integer programming problems, The Computer Journal 8 (1965) pp 250-255.

[16] M. Fitting, *Proof Methods for Modal and Intuitionistic Logics*, Reidel (now Kluwer), 1983, second edition forthcoming.

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

[18] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, *PVM: Parallel Virtual Machine*, MIT Press, 1994.

[19] H. Greenberg, A branch-and-bound solution to the general scheduling problem, Operations

Research 8 (1968) pp 353-361.
[20] D. Greis, *The Science of Programming, Springer-Verlag*, 1981.
[21] T. Griffin, A formulas-as-type notion of control, in *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, 1990.
[22] W. Gropp, E. Lusk and A. Skjellum, Using MPI, *Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 1994.
[23] J. Jaffar and J.-L. Lassez, Constraint logic programming, in *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, 1987.
[24] J. Jaffar and M. Maher, Constraint logic programming: a survey, Journal of Logic Programming 19/20 (1994) pp 503-581.
[25] J. Jaffar, S. Michaylov, Methodology and implementation of a CLP system, n *Proceedings of the 1987 International Logic Programming Conference*, MIT Press, 1987.
[26] J.-L. Krivine, Classical logic, storage operators and second-order lambda-calculus, Annals of Pure and Applied Logic 68 (1994) pp 53-78.
[27] A.H. Land and A.G. Doig, An automatic method of solving discrete programming problems, Econometrica 28 (1960) pp 497-520.
[28] J.-L. Lassez and K. McAloon, A canonical form for generalized linear constraints, Journal of Symbolic Computation 13 (1992) pp 1-24.
[29] K. McAloon and C. Tretkoff, 2LP: Linear Programming and Logic Programming, in *Principles and Practice of Constraint Programming*, V. Saraswat and P. van Hentenryck eds., MIT Press, pp 101-116.
[30] K. McAloon and C. Tretkoff, *Optimization and Computational Logic*, John Wiley and Sons, 1996.
[31] S.S. Nielsen, A C++ class library for mathematical programming, in *The Impact of Emerging Technologies on Computer Science and Operations Research*, edited by S. G. Nash and A. Sofer, Kluwer, 1995, pp 221-243.
[32] J.-F. Puget, A C++ Implementation of CLP, ILOG Technical Report, 1994.
[33] Salkin, H.M., and Mathur, K., *Foundations of Integer Programming*, North-Holland, 1989.
[34] J. Sankaran, *Bidding Systems for Certain Nonmarket Allocations of Indivisible Items*, Thesis, University of Chicago, 1989.
[35] P. van Hentenryck, *Constraint Satisfaction in Logic Programming*, MIT Press, 1989.
[36] D.H.D. Warren, An Abstract Prolog Instruction Set, Technical Report 309, SRI International, 1983.
[37] H. P. Williams, *Model Building in Mathematical Programming*, John Wiley and Sons, 1994.
[38] P. Winston, *Artificial Intelligence*, Addison-Wesley, 1992.