

LOCALIZER: A Modeling Language for Local Search

LAURENT MICHEL AND PASCAL VAN HENTENRYCK / *Brown University, Box 1910, Providence, RI 02912,*
Email: {ldm,pvh}@cs.brown.edu

(Received: October 1997; revised: June 1998; accepted: September 1998)

Local search is a traditional technique to solve combinatorial search problems and has raised much interest in recent years. The design and implementation of local search algorithms is not an easy task in general and may require considerable experimentation and programming effort. However, contrary to global search, little support is available to assist the design and implementation of local search algorithms. This paper is an attempt to support the implementation of local search. It presents the preliminary design of LOCALIZER, a modeling language which makes it possible to express local search algorithms in a notation close to their informal descriptions in scientific papers. Experimental results on our first implementation show the feasibility of the approach.

Most combinatorial search problems are solved through global or local search. In global search, a problem is divided into subproblems until the subproblems are simple enough to be solved directly. In local search (LS), an initial configuration is generated and the algorithm moves from the current configuration to a neighborhood configuration until a solution (decision problems) or a good solution (optimization problems) has been found or the resources available are exhausted. The two approaches have complementary strengths, weaknesses, and application areas. The design of global search algorithms is now supported by a variety of tools, ranging from modeling languages such as AMPL^[2] and NUMERICA^[12] to constraint programming languages such as CHIP, ILOG SOLVER, CLP(\mathcal{R}), PROLOG-IV, and OZ to name only a few. In contrast, little attention has been devoted to the support of LS, despite the increasing interest in these algorithms in recent years. (Note, however, that there are various efforts to integrate local search in CLP languages, e.g., [11]). However, the design of LS algorithms is not an easy task. The same problem can be modeled in many different ways (see for instance [5]), making the design process an inherently experimental enterprise. In addition, efficient implementations of LS algorithms often require maintaining complex data structures incrementally, which is a tedious and error-prone activity.

This paper reports on an attempt to support the design and implementation of LS algorithms. It presents the design of LOCALIZER, a modeling language that makes it possible to describe LS algorithms in a notation close to the informal presentations of these procedures in scientific papers. LOCALIZER statements are organized around the traditional concepts of LS algorithms (e.g., neighborhoods, acceptance criteria, and restarting states), they support the essence of

many LS algorithms (e.g., local improvement,^[8] simulated annealing,^[6] tabu search,^[3] and GSAT^[10], a greedy local search procedure designed to find models for satisfiability problems), and they can be tailored to the application at hand to exploit its underlying structure. The main technical tool in LOCALIZER is the concept of *invariant*, which relieves users from the need of maintaining complex data structures incrementally and makes it simpler to define neighborhoods concisely. As a consequence, LOCALIZER may significantly simplify the implementation of LS algorithms and supports the design process by easing and encouraging the experimental work. The practicability of LOCALIZER is demonstrated by some experimental results showing that it compares well with special-purpose implementations of some LS algorithms.

It is important to stress the difference between LOCALIZER and traditional modeling languages such as AMPL and GAMS. In traditional modeling languages, users mostly focus on describing their problems in a declarative and high-level way and they are relieved from most algorithmic concerns. These languages can in fact be implemented in many ways, using a variety of algorithms from mathematical programming. LOCALIZER, on the hand, is tailored to a class of combinatorial optimization algorithms, namely local search algorithms. Users of LOCALIZER in fact describe, in a high-level way, a local search algorithm. LOCALIZER combines declarative constructs (e.g., invariants) and imperative constructs to formulate these algorithms concisely. Note that a recent proposal along these lines, but for global search, can be found in [1].

The main contribution of this article is to show that local search algorithms can be supported by very high-level modeling languages that shorten their development time substantially while preserving most of the efficiency of special-purpose implementations. This article is a proof of concept: the design of LOCALIZER presented in this paper should not be viewed as a final specification but as a first step towards supporting local search algorithms. There are several extensions that are being contemplated, including additional support for tabu search, the support for genetic algorithms (e.g., maintaining multiple configurations), and the integration of consistency techniques (e.g., arc-consistency).

The rest of this paper aims at giving readers a description of the features of LOCALIZER. Section 1 gives a tour of LOCALIZER. Sections 2, 3, and 4 illustrate LOCALIZER on three

Subject classifications: Modeling language

Other key words: Modeling language, local search, computer science software

```

procedure LOCALIZER
begin
1  s := startState();
2  for search := 1 to MaxSearches while Global Condition do
3    for trial := 1 to MaxTrials while Local Condition do
4      if satisfiable(s) then
5        return s;
6      select n in neighborhood(s);
7      if acceptable(n) then
8        s := n;
9      s := restartState(s);
end

```

Figure 1. The computation model of LOCALIZER.

problems: satisfiability, graph coloring, and graph partitioning. Section 5 contains the experimental results, and Section 6 concludes the paper.

1. A Tour of LOCALIZER

To understand statements in LOCALIZER, it is best to consider first the underlying computational model. Figure 1 depicts the computational model of LOCALIZER for decision problems. The model captures the essence of most local search algorithms. The algorithm performs a number of local searches (up to **MaxSearches** and while a global condition is satisfied). Each local search consists of a number of iterations (up to **MaxTrials** and while a local condition is satisfied). For each iteration, the algorithm first tests if the state is satisfiable, in which case a solution has been found. Otherwise, it selects a candidate move in the neighborhood and moves to this new state if this is acceptable. If no solution is found after **MaxTrials** or when the local condition is false, the algorithm restarts a new local iteration in the state *restartState(s)*. The computation model for optimization problems is similar, except that line 5 needs to update the best solution so far if necessary, e.g., in the case of a minimization,

```

5  if value(s) < bestBound then
5.1  bestBound := value(s);
5.2  best := s;

```

The optimization algorithm of course should initialize **bestBound** properly and return the best solution found at the end of the computation.

The purpose of a LOCALIZER statement is to specify, for the problem at hand, the instance data, the state, and the generic parts of the computation model (e.g., the neighborhood and the acceptance criterion). A LOCALIZER statement consists of a number of sections as depicted in Figure 2. The instance data is defined by sections **DataType**, **Constant**, and **Init**, using traditional data structures from programming languages. The state is defined as the values of the variables. The neighborhood is defined in the **Neighborhood** section, using objects from previous sections. The acceptance criterion is part of the definition of the neighborhood. The initial state is defined in section **Start**. The restarting states are defined in section **Restart**, the parameters (e.g. **MaxTrials**) are given in the **Parameter** section, and the global and local

```

⟨ Model ⟩ ::= [solve | optimize]
[⟨DataType⟩]
[⟨Constant⟩]
⟨Variable⟩
[⟨Invariant⟩]
[⟨Operator⟩]
⟨Satisfaction⟩
[⟨Objective Function⟩]
⟨Neighborhood⟩
[⟨Start⟩]
[⟨Restart⟩]
[⟨Parameter⟩]
[⟨Global Condition⟩]
[⟨Local Condition⟩]
[⟨Init⟩]

```

Figure 2. The structure of LOCALIZER statements.

conditions are given in sections **Global Condition** and **Local Condition**. Note that all the identifiers in boldface in the computation model are in fact keywords of LOCALIZER.

As mentioned previously, the most original aspects of LOCALIZER are in the specifications of the neighborhood and the acceptance criterion. Of course, some of the notations are reminiscent of languages such as AMPL and CLAIRE at the syntactical level but the underlying concepts are fundamentally different. In the rest of this section, we describe the most original aspect of LOCALIZER without trying to be comprehensive.

1.1 Neighborhoods

Neighborhoods in LOCALIZER are defined through basic instructions of the form

```

move op(x1, . . . , xn)
where
  x1 from S1;
  . . .
  xn from Sn
[accept when ⟨AcceptanceCriterion⟩]

```

where *op* is an operator or a sequence of instructions using traditional programming language constructs. Assuming that *s* is the current configuration and *Post(s, i)* represents the configuration obtained by executing *i* in *s*, the instruction defines the neighborhood

$$\{Post(s, op(x_1, \dots, x_n)) \mid x_1 \in S_1 \ \& \ \dots \ \& \ x_n \in S_n\},$$

selects one of its elements, and checks if it satisfies the acceptance criterion if any. Neighborhoods in LOCALIZER are also defined through basic instructions of the form

```

best move op(x1, . . . , xn)
where
  x1 from S1;
  . . .
  xn from Sn
[accept when ⟨AcceptanceCriterion⟩]

```

The instruction defines the same neighborhood as before but selects the move *m* with the best value of the objective

function in $Post(s, m)$. This element is then tested against the acceptance criterion, if any. The sets in the **where** statement are specified using the syntax

```

⟨Select Set⟩ ::= ⟨Set Expr⟩
              ::= ⟨Set Expr⟩ minimizing ⟨Expr⟩
              ::= ⟨Set Expr⟩ maximizing ⟨Expr⟩

```

For instance, the excerpt

```

best move
  a[i] := !a[i]
where
  i from {1..n}
accept when . . .

```

may be used to specify the flipping strategy of GSAT, while an instruction of the form

```

move
  q[i] := v
where
  i from Conflicts;
  v from {1..n} minimizing
  sizeof({j: integer|select j from 1..n where q[j] = v or
  q[j] = v + i - j or q[j] = v + j - i})
accept when . . .

```

is an example of the min-conflict heuristics of [7]. The GSAT statement simply specifies to flip the value of the literal $a[i]$ ($1 \leq i \leq n$), which produces the best improvement in the objective function (keyword **best**). The min-conflict heuristics selects a conflicting variable (e.g., a queen being attacked) and chooses a new value that minimizes the number of conflicts for the variable. Note that best improvement can be easily combined with the min-conflict heuristics simply by adding the keyword **best** before **move**.

Note also that, in addition to the set-selection definitions, LOCALIZER makes it possible to specify constants in the **where** part of the neighborhood as in

```

move
  x[i] := c
where
  i from {1..n};
  c from Candidates;
  nb = sum(j in C[c]) A[i, j];
  ob = sum(j in C[x[i]]) A[i, j];
  d = diff(x[i], -1, -ob) + diff(c, +1, +nb);
accept when . . .

```

which is an excerpt of the graph-coloring model presented later in this paper. Here, nb , ob , and d are constants that are computed in the **where** part: these constants will be used in the acceptance criterion to decide whether to take the move.

The **move** instructions can also be composed to select a move from several neighborhoods using the **try** instruction:

```

try
  [Probability:](Move Statement);
  . . .
  [Probability:](Move Statement);
end

```

In this case, LOCALIZER considers each **move** statement in sequence until one is successful or all have been considered. The **move** statements can be conditional to a probability, in which case LOCALIZER considers them with the given probability. The **try** instruction is useful, for instance, to implement the random walk/noise strategy, as described for GSAT for instance,

```

try
  0.1:
    move
      a[i] := !a[i]
    where
      i from OccurInUnsatClause
    accept when always . . . ;
  default:
    best move
      a[i] := !a[i]
    where
      i from {1..n}
    accept when . . . ;
end

```

Here, LOCALIZER flips an arbitrary variable in an unsatisfied clause with a probability of 0.1 and applies the standard strategy with a probability of 0.9. Note that LOCALIZER simply goes to the next iteration if the selected neighborhood is empty, because other neighborhoods may be non-empty.

1.2 Invariants

The specification of the set expressions S_1, \dots, S_n in the **move** instructions is probably the most interesting part of the neighborhood definitions. These sets are generally defined in terms of invariants, one of the main concepts of LOCALIZER to simplify the implementation of LS algorithms. Informally speaking, an invariant is an expression of the form $v: t = exp$ and LOCALIZER guarantees that, at any time during the computation, the value of variable v of type t is the value of the expression exp (also of type t of course). Typical examples of invariants are

```

v:integer = sum(i in S) a[i];
C:{integer} = {i:integer|select i from S where A[i] = 0};
D[i in I]:{integer} = {j:integer|select j from S where a[j] = i};

```

The first invariant specifies that v (an integer) is the summation of the $a[i]$ ($i \in S$), the second invariant specifies that C (a set of integers) is the set of i such that $a[i] = 0$, and the last invariant specifies that $D[i]$ ($i \in I$) is the set of all j in S such that $a[j] = i$. More generally, invariants are specified by expressions using standard arithmetic operators, boolean connectives, aggregate operators such as summation, product, maximum, minimum, and sizeof, conditional expressions, explicit sets, sets defined implicitly using expressions, and set union, intersection, and difference. Invariants can also be defined over complex data structures. For instance, the following excerpt from a graph-coloring model to be presented later

```

Optimize
Data Type:

```

```

edge = record
  s:integer;
  t:integer;
end;
...
Invariant:
...
...
  B[k in 1..n]:{edge} = {{i, j}:edge|
    select i from C[k] &
    select j from C[k] where A[i, j]};
...

```

defines $B[k]$ as the set of edges whose vertices are in color class k . The set $B[k]$ is computed by considering two vertices in color class k and checking if they are adjacent. Finally, it is useful to mention that LOCALIZER offers a predefined invariant $distribute(x, c, S)$ defined as

```

c[i in I]:{integer} = {j:integer|select j from S where x[j] =
  i};

```

because of the ubiquity of this invariant in practical applications.

LOCALIZER uses efficient incremental algorithms to maintain these invariants during the computation, automating one of the tedious and time-consuming tasks of LS algorithms. For instance, in the first three invariants depicted previously, whenever a value $a[k]$ is changed, v , C , and $D[j]$ are updated in constant time in our current implementation. Similarly, when a value in x is changed in $distribute(x, c, S)$, c is also updated in constant time.

As a consequence, invariants make it possible to specify **what** needs to be maintained incrementally without considering **how** to do so.

1.3 Acceptance Criterion

Once an element of the neighborhood has been selected, LOCALIZER determines if it is an appropriate move. The acceptance criterion is built using the syntax

```

⟨AcceptanceCriterion⟩ ::= ⟨AcceptanceStatement⟩
  ::= in resulting state ⟨AcceptanceStatement⟩
  ::= in current state ⟨AcceptanceStatement⟩
⟨AcceptanceStatement⟩ ::= ⟨AcceptanceCondition⟩
  ::= ⟨AcceptanceCondition⟩ 3
  ⟨Statement⟩
⟨AcceptanceCondition⟩ ::= always
  ::= improvement
  ::= noDecrease
  ::= ⟨Expr⟩
  ::= ⟨AcceptanceCondition⟩ and
  ⟨AcceptanceCondition⟩
  ::= ⟨AcceptanceCondition⟩ or ⟨AcceptanceCondition⟩
  ::= not ⟨AcceptanceCondition⟩

```

There are several orthogonal concepts to be discussed here and we will describe them separately. The main part of the acceptance criterion is the acceptance condition. Condition

always accepts all moves, condition **improvement** accepts a move only if it improves the value of the objective function, and condition **noDecrease** accepts a move only if it does not degrade the value of the objective function. The acceptance criterion can also be expressed using a Boolean expression involving the objects defined by the invariants. All these basic cases can be combined with Boolean connectives. For instance, a local improvement strategy is specified as

```

move
...
accept when improvement;

```

while a simple tabu-search strategy may be specified using a statement of the form

```

move ...
where o from S
...
accept when
  o notin Tabu;

```

When using a Boolean expression, it is important to specify in which state to evaluate the expression. Generally, it is simpler to design models using the state after the move (called the *resulting* state), but more efficient to design models using the state before the move (called the *current* state) to avoid having to simulate the move. For instance, the excerpt

```

move x[i] := c
where
  i from {1..n};
  c from C
accept when in current state
  d[i] > 0;

```

accepts a move when $d[i]$ is positive in the state before the move. When **in current state** is not specified, the default **in resulting state** is assumed. Acceptance instructions can also be combined using an instruction of the form

```

⟨AcceptanceCriterion⟩ ::=
  [Probability:]⟨AcceptanceStatement⟩
cor
...
cor
  [Probability:]⟨AcceptanceStatement⟩;

```

which tries each acceptance criterion in sequence until one has succeeded or all have failed. Note that keyword **cor** stands for *conditional or*. A criterion may be associated with a probability, in which case LOCALIZER considers the statement with the given probability. To specify the probability easily, LOCALIZER also provides, as a keyword **delta**, the variation of the objective function produced by the move. Note that **delta** only makes sense in the context of the resulting state as do **improvement** and **noDecrease**. For instance, a typical simulated annealing procedure uses the acceptance criterion

```

accept when
  improvement

```

```

Solve
Data Type:
  clause = record
    p : {integer};
    n : {integer};
  end;
Constant:
  m: integer = ...;
  n: integer = ...;
  cl: array[1..m] of clause = ...;
Variable:
  a: array[1..n] of boolean;
Invariant:
  nbtl[ i in 1..m ] : integer = sum(j in cl[i].p) a[j] + sum(j in cl[i].n) !a[j];
  nbClauseSat : integer = sum(i in 1..m) (nbtl[i] > 0);
Satisfiable:
  nbClauseSat = m;
Objective Function:
  maximize nbClauseSat;
Neighborhood:
  best move a[i] := !a[i]
  where i from {1..n}
  accept when noDecrease;
Start:
  forall(i in 1..n)
    random(a[i]);
Restart:
  forall(i in 1..n)
    random(a[i]);

```

Figure 3. A simple model of GSAT.

cor
 $\Pr(e^{-\text{delta}/t})$: always;

where t is the temperature. This criterion accepts a move if it improves the value of the objective function or, if not, with a probability $e^{-\text{delta}/t}$.

The acceptance statement makes it possible to associate an action with each acceptance criterion, which is useful to express some termination condition. For instance, to implement a strategy that terminates when the objective function has been stable for a number of iterations, one would write:

```

Neighborhood:
  move ...
  accept when
    improvement  $\exists$  nbStableIter := 0
cor
  noDecrease  $\exists$  nbStableIter++;
Local Condition:
  nbStableIter  $\leq$  maxStableIter;

```

2. Satisfiability

We now illustrate LOCALIZER on a number of applications. Our first application is satisfiability (SAT), and we illustrate how GSAT^[10] can be expressed in LOCALIZER. GSAT illustrates many aspects of LOCALIZER as well as several interesting modeling issues.

Problems in SAT are described in terms of a number of clauses, each clause consisting of a number of literals. As is

traditional, a literal is simply an atom (positive atom) or the negation of an atom (negative atom). The goal is to find an assignment of Boolean values to the atoms such that all clauses are satisfied, a clause being satisfied if one of its positive atoms is true or one of its negative atoms is false. The basic idea of GSAT is to start from a random assignment of Boolean values and to select the atom that, when its value is inverted, produces a state with the largest number of clauses satisfied. Note that GSAT accepts only moves that do not decrease the number of clauses satisfied.

2.1 A Simple Model of GSAT

Figure 3 depicts a simple model of GSAT.

Instance Representation.

In the model, atoms are represented by integers from 1 to n and a clause is represented by two sets: the set of its positive atoms p and the set of its negative atoms n . A SAT problem is simply an array of m clauses. The actual instance is described in the **Init** section, which is not shown.

State Definition.

The state is specified by the truth values of the atoms and is captured in the array a , where $a[i]$ represents the truth value of atom i .

```

Solve
Data Type:
  clause = record
    p : {integer};
    n : {integer};
  end;
Constant:
  m: integer = ...;
  n: integer = ...;
  cl: array[1..m] of clause = ...;
  po: array[ i in 1..n ] of {integer} :=
    { c : integer | select c from 1..m where i in cl[c].p };
  no: array[ i in 1..n ] of {integer} :=
    { c : integer | select c from 1..m where i in cl[c].n };
Variable:
  a: array[1..n] of boolean;
Invariant:
  nbtI[ i in 1..m ] : integer = sum(j in cl[i].p) a[j] + sum(j in cl[i].n) !a[j];
  g01[ i in 1..n ] : integer = sum(j in po[i]) (a[j] = 0) - sum(j in no[i]) (a[j] = 1);
  g10[ i in 1..n ] : integer = sum(j in no[i]) (a[j] = 0) - sum(j in po[i]) (a[j] = 1);
  gain[ i in 1..n ] : integer = if a[i] then g10[i] else g01[i];
  maxGain : integer = max(i in 1..n) gain[i];
  Candidates : {integer} =
    { i : integer | select i from 1..n where gain[i] = maxGain and gain[i] ≥ 0 };
  nbClauseSat : integer = sum(i in 1..m) (nbtI[i] > 0);
Satisfiable:
  nbClauseSat = m;
Neighborhood:
  move a[i] := !a[i]
  where i from Candidates;
Start:
  forall(i in 1..n)
    random(a[i]);
Restart:
  forall(i in 1..n)
    random(a[i]);

```

Figure 4. A more incremental model of GSAT.

Neighborhood.

The neighborhood uses two invariants: the number of true literals for clause i , denoted by $nbtI[i]$, and the number of clauses satisfied, denoted by $nbClauseSat$. $nbtI[i]$ is computed by counting the number of positive atoms and the negation of the negative atoms. $nbClauseSat$ is the number of satisfied clauses. LOCALIZER maintains these invariants incrementally: in particular, each time a variable $a[j]$ is changed, the invariants are recomputed in time linearly proportional to the number of occurrences of j . The neighborhood is then defined by specifying that LOCALIZER must flip the value of the atom that produces a state maximizing the number of clauses satisfied, according to the description of GSAT. The acceptance criterion specifies that the new state should not decrease the number of clauses satisfied.

The satisfiability section specifies that the state is a solution when all clauses are satisfied, which can be stated simply in terms of the available data items. The objective function simply maximizes the number of clauses satisfied. The initial state consists of a random assignment of the variables.

It is useful at this point to step back and to look at the

simplicity of the model. It is difficult in fact to imagine a more compact definition, and invariants clearly play an important role in the model simplicity.

2.2 A More Incremental Model of GSAT

As mentioned previously, the same problem can often be expressed in many different ways. In this section, we study a more incremental version of GSAT, which is depicted in Figure 4. The interest of the new model for this paper lies in the illustration of several interesting features, which we now describe.

The simple model is incremental in the computation of the invariants but it does not maintain the set of candidates for flipping incrementally: the candidates are obtained by evaluating the number of clauses satisfied in the new state obtained by flipping each variable. The new model is completely incremental and maintains the set of candidates for flipping at any computation step. This new model is significantly faster than the simple model, while remaining easy to design. Note that LOCALIZER cannot in general deduce such optimizations automatically, given the fact that operators may be arbitrarily complex and that some problems are not easily amenable to such optimizations.

The Model.

The representation is the same as in the simple model. However, the **Constant** section contains two additional sets: $po[i]$ represents the set of clauses in which atom i appears positively, while $no[i]$ represents the set of clauses in which atom i appears negatively.

Neighborhood.

The invariants are more involved in this model and they maintain incrementally the set of candidates which can be selected for a flip. The informal meanings of the new invariants are the following. $g01[i]$ represents the change in satisfied clauses when changing the value of atom i from false to true, assuming that atom i is currently false. Obviously, the flip produces a gain for all unsatisfied clauses where atom i appears positively. It also produces a loss for all clauses where i appears negatively and is the only atom responsible for the satisfaction of the clause. $g10[i]$ represents the change in satisfied clauses when changing the value of atom i from true to false, assuming that atom i is currently true. It is computed in a way similar to $g01$. $gain[i]$ represents the change in satisfied clauses when changing the value of atom i . It is implemented using a conditional expression in terms of $g01[i]$, $g10[i]$, and the current value of atom i . $maxGain$ is simply the maximum of all gains. Finally, *Candidates* describes the set of candidates for flipping. It is defined as the set of atoms whose gain is positive and maximal. Once the invariants have been described, the neighborhood is defined by flipping one of the candidates. There is no need to specify an optimization qualifier, this information is already expressed in the invariants. Note that some invariants in this model involve sets, conditional expressions, and aggregation operators that are maintained incrementally. They clearly illustrate the significant support provided by LOCALIZER. Users can focus on describing the data needed for their application, while LOCALIZER takes care of maintaining these data efficiently.

Adding Weights.

Reference [9] proposes to handle the special structure of some SAT problems by associating weights to the clauses and updating these weights each time a new local search is initiated. We now show how easy it is to integrate this feature. The changes consist in introducing weight variables $w[i]$ in the state, in modifying the computations of the invariants for $g01$ and for $g10$ by multiplying the appropriate terms by the weights, i.e.,

$$g01[i \text{ in } 1..n]: \text{integer} := \text{sum}(j \text{ in } po[i]) w[j] \times (a[j] = 0) - \text{sum}(j \text{ in } no[i]) w[j] \times (a[j] = 1);$$

$$g10[i \text{ in } 1..n]: \text{integer} := \text{sum}(j \text{ in } no[i]) w[j] \times (a[j] = 0) - \text{sum}(j \text{ in } po[i]) w[j] \times (a[j] = 1);$$

and in updating the weights after each local search by changing the restarting section to

Restart:

```
forall(i in 1..m)
  w[i] := w[i] + (nbt[i] = 0);
forall(i in 1..n)
  random(a[i]);
```

The rest of the statement remains exactly the same, showing the ease of modification of LOCALIZER statements.

3. Graph Coloring

This section considers the graph-coloring problem, i.e., the problem of finding the smallest number of colors to label a graph such that any two adjacent vertices have a different color. It shows how a simulated annealing algorithm proposed in [5] can be expressed in LOCALIZER. Of particular interest is once again the close similarity between the problem description and the model. In addition, graph coloring makes it possible to discuss some interesting issues about the tradeoff between expressiveness and efficiency. Note finally that graph coloring could be expressed as an instance of SAT as could any NP-Complete problem. However, it is often desirable to specialize the local search to the problem at hand, and LOCALIZER makes it possible to exploit the special structure of each problem.

For a graph with n vertices, the algorithm considers n colors that are the integers between 1 and n . Color class C_i is the set of all vertices colored with i , and the bad edges of C_i , denoted by B_i , are the edges whose vertices are both colored with i . The main idea of the algorithm is to minimize the objective function $\sum_{i=1}^n 2|B_i||C_i| - |C_i|^2$. This function is interesting because its local minima are valid colorings. To minimize the function, the algorithm chooses a vertex and chooses a color whose color class is non-empty or one of the unused colors. It is important to consider only one of the unused colors to avoid a bias towards unused colors. A move is accepted if it improves the value of the objective function or, if not, with a standard probability of simulated annealing algorithms. Figure 5 depicts the simulated annealing model for graph coloring. The model closely follows the above description and adds standard simulated annealing techniques.

3.1 A Simple Model of Graph Coloring

The instance data is described by the number of vertices n (each vertex being a number between 1 and n), the set of edges E between vertices, and the annealing parameters $cutOff$, $chPerc$, and $maxFreeze$, which are described subsequently. The adjacency matrix A is computed automatically from the edges. The state is represented by the variables $x[i]$, which represent the colors of vertices, by the temperature t , and by two other annealing parameters, fc and ch , which are used to control the local and global conditions. The invariants describe the sets C_i , B_i , and the objective function. The invariant $\text{distribute}(x, C, 1..n)$ computes the sets C_i . The “unused” color is obtained by taking the smallest unused color, i.e., the smallest color whose class is empty. The set of candidate colors are thus all the “used” colors together with the selected “unused” colors. The bad edges are maintained for each color class by considering adjacent vertices in the color class. The total number of bad edges ($countB$) is also

```

Optimize
Data Type:
  edge = record s : integer; t : integer; end;
Constant:
  n      : integer = ...;
  E      : {edge} = ...;
  cutOff : real = ...;
  chPerc : real = ...;
  maxFreeze: real = ...;
  A      : array[i in 1..n, j in 1..n] of boolean := ⟨i, j⟩ in E;
Variable:
  x : array[1..n] of integer;
  t : integer;
  fc : integer;
  ch : integer;
Invariant:
  distribute(x, C, 1..n);
  Empty : {integer} = { i : integer | select i from 1..n where size(C[i]) = 0 };
  NEmpty : {integer} = { i : integer | select i from 1..n where size(C[i]) > 0 };
  unused : integer = minof(Empty);
  Candidates : {integer} = NEmpty union unused;
  B[k in 1..n] : {edge} = {⟨i, j⟩ : edge | select i from C[k] & select j from C[k] where A[i, j] };
  f : integer = sum(i in 1..n) (2×size(C[i])×size(B[i]) - size(C[i])2)
  countB : integer = sum(i in 1..n) size(B[i]);
Satisfiable:
  countB = 0;
Objective Function:
  minimize f;
Neighborhood:
  move x[i] := c
  where
    i from {1..n};
    c from Candidates
  accept when
    improvement → { if countB = 0 then fc = 0 endif; ch:=ch+1; }
  cor
    noDecrease
  cor
    Pr( $e^{-\text{delta}/t}$ ) : always → ch:=ch+1;
Start:
  T := initTemp; fc:=0; ch:=0; forall(i in 1..n) random(x[i]);
Restart:
  T := factor × T; if ch/trial < chPerc then fc:=fc+1 endif;
Parameter: MaxTrials := 90 * sf * n;
Local Condition: ch < round(cutOff * n);
Global Condition: fc < maxFreeze;

```

Figure 5. A graph coloring model.

maintained to decide satisfiability. The neighborhood is then described in a simple way by choosing a vertex and a candidate color. Acceptance obeys the standard simulated annealing criterion (as was already shown in Section 2). The stopping criteria are directly derived from the algorithm description in [5]. The inner local search can only perform $\text{round}(\text{cutOff} * n)$ variations of the objective function, and the variable ch maintains the number of variations by incrementing ch in the acceptance actions. The motivation behind this choice is to control the search when the temperature is high. The global iteration is stopped when $fc \geq \text{maxFreeze}$. The intuition here is that fc represents the number of local searches without significant progress. fc is reset to 0 whenever there is an improvement. It is incremented each time

the local search is restarted and there was no significant change in the previous local search.

3.2 A More Incremental Model of Graph Coloring

The above model uses the resulting state in the acceptance criterion. This means that, in order to evaluate if a move is acceptable, it is necessary to simulate the move, e.g., to update all the invariants and to undo the changes if the move is not acceptable. Although the algorithms in LOCALIZER are incremental, such a simulation may become the most consuming part of the algorithm at low temperatures when many candidate moves are discarded.

However, it is often possible to improve a model by defining the acceptance criterion in the current state. Once

```

Operator:
integer f(C : integer,S : integer) { return 2 * C * S - C2; }
integer diff(i : integer,dc : integer,ds : integer) {
  { return f(size(C[i])-dc,size(B[i])-ds) - f(size(C[i]),size(B[i])); }
Neighborhood:
move x[i] := c
where
  i from {1..n};
  c from Candidates;
  nb = sum(j in C[c]) A[i, j];
  ob = sum(j in C[x[i]]) A[i, j];
  d = diff(x[i],-1,-ob) + diff(c,1,nb)
accept when in current state
  d < 0 → { if countB = 0 then fc = 0 endif; ch:=ch+1; }
cor
  d=0
cor
  Pr(e-d/t): always → ch:=ch+1;

```

Figure 6. A more incremental neighborhood for graph coloring.

again, this process is difficult to automate in general, because it may involve symbolic manipulations and/or some semantic aspects of the problem (e.g., whether a matrix is symmetric or not). Figure 6 depicts a new neighborhood definition to make the model more incremental. The key insight here is that it is possible to evaluate the impact of the move in the current state by simply looking at the bad edges in color classes $C[x[i]]$ and $C[c]$ and deducing the variation d of the objective function. In particular, the old class color of vertex i , i.e., $x[i]$, decreases by one in size, while its number of bad edges decreases by

$$\text{sum}(j \text{ in } C[x[i]]) A[i, j].$$

Similarly, the new class color c of vertex i increases by one in size and its number of bad edges increases by

$$\text{sum}(j \text{ in } C[c]) A[i, j].$$

The variation d of the objective function is computed in the neighborhood definition, and, in the acceptance criterion, keywords **improvement**, **noDecrease**, and **delta** are replaced by $d < 0$, $d = 0$, and d , respectively. The performance gain on this problem is significant (about a factor of 5 on large instances (500 vertices)).

4. Graph Partitioning

This section considers the graph-partitioning problem, i.e., the problem of finding a partition of the vertices of a graph into two sets of equal size that minimize the number of edges connecting the two sets. It describes a LOCALIZER model for the simulated annealing algorithm presented in [4]. Traditional local search for graph partitioning^[8] maintains two sets of equal size and swaps vertices between these two sets. The model of [4] relaxes this idea of maintaining a feasible solution and a move consists of selecting a vertex and moving it to the other set. The objective function combines the objective of minimizing the connections between the two sets with the desire to favor balanced solutions and is given as

$$SB + \alpha * IMB^2$$

where SB is the number of connections, IMB is the imbalance between the two sets, and α is a parameter of the algorithm.

The LOCALIZER model is depicted in Figure 7. The instance data is described by the number of vertices n (each vertex being a number between 1 and n), the set of edges E between vertices, and the annealing parameters (which we will not describe for this model because they are again taken directly from [4]). The adjacency matrix A is computed automatically from the edges. The state is represented by the variables $x[i]$ and some annealing variables. Variable $x[i]$ is true if i belongs to set S_0 and false otherwise. The invariants $P0S$ and $P1S$ maintain the size of the sets S_0 and S_1 , respectively. Invariants $EX[i]$ represents the traditional internal cost of a vertex, i.e., the number of edges connecting the vertex to vertices in the other set. IMB is the imbalance between the two sets, and OBJ is the objective function mentioned earlier. The neighborhood is essentially similar to the neighborhood of the graph-coloring problem, except that here a candidate move consists of flipping the value of a variable.

Once again, there is a small distance between the model and its statement in LOCALIZER. As was true for graph coloring, it is easy to find a more incremental version of the model by evaluating the variation of the objective function in the current state using the variations on the internal and external costs.

5. Experimental Results

This section describes some preliminary results on the implementation of LOCALIZER (about 25,000 lines of C++). The goal is not to report the final word on the implementation but rather to suggest that LOCALIZER can be implemented with an efficiency comparable to specific local search algorithms. To demonstrate practicability, we experimented with LOCALIZER on the problems described in this article: SAT, graph coloring, and graph partitioning.

```

Optimize
Type:
  edge = record s : integer; t : integer; end;
Constant:
  n : integer = ...;
  E : {edge} = ...;
  alpha : real = ...;
  cutOff : real = ...;
  sf : integer = ...;
  chPerc : integer = ...;
  A : array[i in 1..n] of {integer} :=
    {j : integer | select j from 1..n where (i, j) in E or (j, i) in E };
Variable:
  x : array[1..n] of boolean;
  t : real;
  fc : integer;
  ch : integer;
Invariant:
  POS : integer = sum(i in 1..n) x[i];
  PIS : integer = n - POS;
  EX[i in 1..n] : integer = sum(k in A[i]) (x[k] <> x[i])/2;
  SB : integer = (sum(i in 1..n) EX[i])/2;
  IMB : integer = POS - PIS;
  OBJ : real = SB + alpha * IMB2;
Satisfiable: IMB=0;
Objective Function: minimize OBJ;
Neighborhood:
  move x[i] := !x[i]
  where
    i from {1..n};
  accept when
    improvement → ch:=ch+1
  cor
  noDecrease
  cor
  Pr(e-delta/t) : always → ch:=ch+1;
Start:
  t:=10;fc:=0;ch:=0;
  forall(i in 1..n) x[i] := random({true,false});
Restart:
  t := t * 0.95;
  if ch * 100/trial < chPerc then fc := fc + 1 endif;
  ch := 0;
Parameter: MaxTrials := sf * n;
Local Condition: ch < cutOff * n;
Global Condition: fc < 5;

```

Figure 7. A graph partitioning model.

5.1 GSAT

The GSAT implementation of [10], referred to as the AT&T implementation hereafter, is generally recognized as a fast and very well-implemented system. The experiments were carried out as specified in [10]. Table I gives the number of variables (V), the number of clauses (C), and **MaxTrials** (I) for each class of benchmarks as well as the CPU times in seconds of LOCALIZER (L), the CPU times in seconds of the AT&T implementation (G) as reported in [10], and the ratio L/G . The times of the AT&T implementation are given on a SGI Challenge with a 70 MHz MIPS R4400 processor. The times of LOCALIZER were obtained on a SUN SPARC-10 40 MHz and scaled up by a factor 1.5 to account for the speed difference between the two machines. LOCALIZER times are for the incremental model presented in Section 3. Note that this comparison is not perfect (e.g., the randomization may be different), but it is sufficient for showing that LOCALIZER can be implemented efficiently.

Table I. GSAT: Experimental Results

	V	C	I	L	G	L/G
1	100	430	500	19.54	6.00	3.26
2	120	516	600	40.73	14.00	2.91
3	140	602	700	54.64	14.00	3.90
4	150	645	1500	154.68	45.00	3.44
5	200	860	2000	873.11	168.00	5.20
6	250	1062	2500	823.06	246.00	3.35
7	300	1275	6000	1173.78	720.00	1.63

As can be seen, the AT&T implementation and LOCALIZER behave consistently. The distribution of ratios as problem size increase remains relatively uniform.

The gap between the two systems is about one machine generation (i.e., on modern workstations, LOCALIZER runs as efficiently as the AT&T implementation on machines of three years ago), which is really acceptable given the preliminary nature of our (unoptimized) implementation.

5.2 Graph Coloring

Graph coloring was the object of an extensive experimental evaluation in [5], and this section reports on experimental results along the same lines. The experiments were conducted on graphs of densities 10, 50, and 90 and of sizes 125, 250, and 500. They were also conducted on so-called “cooked” graphs. Cooked graphs have a well-known optimal coloring. A cooked graph with n vertices and with a chromatic number κ is constructed as follows:

1. Randomly assign the vertices with equal probability to κ color classes.
2. For each pair (u, v) of vertices in different color classes, place an edge to connect them with a probability $\kappa/2(\kappa - 1)$.
3. Pick κ vertices, one from each class, and make sure they form a clique in G .

Because of the nature of the experimental results reported in [5], it is not easy to compare the efficiency of LOCALIZER to the efficiency of their algorithm. As a consequence, we decided to build a very efficient C implementation of their algorithm from scratch and to compare it with LOCALIZER. This implementation was performed by a graduate student not connected to the LOCALIZER project. This student was closely supervised to obtain a very efficient incremental algorithm. As far as we can judge, the timings and the quality of this algorithm seem consistent with those in [5]. In the following, we discuss the development time of the two implementations, the quality of the solutions obtained (to make sure that the algorithms are comparable in quality), and the efficiency.

Development Time.

The C implementation of the algorithm is about 1500 lines long and required a full week to develop. This should be

Table II. Graph Coloring: Quality of the Solutions

	Vertices	Density	Size Factor (<i>sf</i>)	Colors	<i>L</i>	<i>C</i>
random	125	50	3	19	92	87
random	250	50	4	20	8	13
				≤ 32	8	6
				33	43	41
				34	44	50
random	500	50	4	≥ 35	4	3
				55	4	8
				56	54	48
				57	41	36
				≥ 58	1	8
				6	48	43
				7	52	54
				8	0	3
random	250	10	1	9	27	29
				10	70	70
				11	3	1
random	500	10	2	15	1	3
				16	79	86
				17	20	11
random	125	90	1	≤ 43	3	0
	125	90	1	44	4	15
				45	30	32
				46	30	26
				≥ 47	33	12
random	250	90	1	≤ 78	4	4
				79	15	19
				80	35	25
				≥ 81	46	52
				≤ 143	30	15
random	500	90	1	144	22	11
				≥ 145	48	38
				<i>Invalid</i>	0	36
cooked	125		4	9	100	100
cooked	250		1	15	100	100
cooked	500		2	25	71	65
				≥ 25	29	35

compared with the concise model presented earlier in this article.

Quality of the Solutions.

Table II describes the quality of the coloring found by LOCALIZER. These results agree with those of the C implementation and with those reported in [5]. Each set of rows corresponds to a class of graphs and to 100 executions of LOCALIZER on graphs from this class. The rows in each set

report on the various values found by LOCALIZER on these graphs and their frequencies. The columns report the number of vertices, the density of the graph, the size factor *sf* used in the experiments, the number of colors found by some solution, and the frequency of colorings of this quality for LOCALIZER (*L*) and the C implementation (*C*). For instance, the first set of rows reports that, on graphs of 125 vertices and density of 50%, 92% of the LOCALIZER executions led to a coloring with 19 colors and 8% of the executions led to a coloring with 20 colors, while the C implementation found 19 colors in 87% of the executions and 20 colors in 13% of the runs. The results are given both for random and cooked graphs and it can be observed that the frequencies are similar for LOCALIZER and the C implementation.

Efficiency.

Table III compares the efficiency of LOCALIZER with the C implementation on the same problems. Each row reports the average time of the two implementations for the 100 graphs in each class and computes the slowdown of LOCALIZER. The experiments were performed on a SUN Sparc Ultra-1 running Solaris 5.5.1 and the standard C++ compiler. Once again, it is useful to observe that both implementations behave rather consistently as program size increases. On these problems, the slowdowns are, in general, slightly higher than a machine generation but they remain reasonable given the preliminary nature of the implementation. This slowdown should also be contrasted with the substantial reduction in development time.

5.3 Graph Partitioning

The problem has been studied experimentally in [4], and, once again, the experiments reported here are based on a similar setting. Two classes of graphs are considered: randomly generated graphs of various density and so-called *geometric* graphs. Geometric graphs are constructed as follows. Pick $2n$ numbers between 0 and 1. Interpret these numbers as the coordinates of vertices lying inside a unit square. Define an edge $\langle i, j \rangle$ between vertex i and j if the Euclidean distance between vertex i and j is less than or equal to a constant d . The average degree of vertices not too close to the boundary of the unit square have an average degree equal to $n\pi d^2$. The instances used in the experiments are generated by choosing a value n and a value for $n\pi d^2$.

Quality of the Solutions.

Table IV depicts the experimental results of LOCALIZER. The first row gives the setting of our parameters: T is the starting temperature, TF is the percentage of reduction of the temperature, SF is the size factor, and the remaining two were described previously. The table reports both the quality of the results and their distribution, as well as the performance of LOCALIZER on these problems. Once again, each row of the table reports the result for 100 executions of LOCALIZER. The columns under *SB* give ranges for the solutions produced by

Table III. Graph Coloring: Efficiency of LOCALIZER

	Vertices	Density	Size Factor (sf)	LOCALIZER (L)	C Implementation (C)	L/C
random	125	50	3	78.3	18.9	4.50
random	250	50	4	82.8	18.4	4.50
random	500	50	4	633.7	123.4	5.10
random	125	10	1	22.5	4.8	4.60
random	250	10	1	109.2	22.02	4.96
random	500	10	2	159.8	28.8	5.50
random	125	90	1	16.18	4.53	3.56
random	250	90	1	49.32	8.89	5.54
random	500	90	1	162.7	29.6	4.88
cooked	125		4	22.09	4.18	5.28
cooked	250		1	37.22	7.79	4.77
cooked	500		2	240.3	49.9	4.80

Table IV. Graph Partitioning: Experimental Results

Graph			Results*						
Class	Vertices	Density or $n\pi d^{\dagger}$	SB			Freq.			LOCALIZER
random	124	2	11-13	14-16	17-19	33	38	29	2.22
random		4	55-59	60-65	66-77	30	34	36	2.40
random		8	159-174	175-190	191-	23	53	24	3.24
random		16	481-560	561-640	641-	36	45	19	4.05
random	250	1	20-24	25-29	30-35	19	22	59	4.68
random		2	92-106	107-121	122-131	12	42	36	5.10
random		4	324-343	344-363	364-380	38	43	19	6.50
random		8	828-877	878-927	928-	37	40	23	9.98
random	500	0.5	48-54	55-59	60-66	15	43	42	10.08
random		1	219-231	232-244	245-256	17	52	31	10.76
random		2	637-661	662-686	686-718	10	15	75	14.44
random		4	1661-1701	1702-1741	1742-1824	22	58	20	20.67
random	1000	0.25	90-103	104-118	119-126	41	52	7	19.99
random		0.5	439-455	456-475	476-503	36	43	21	22.62
random		1	1326-1357	1358-1397	1398-1427	33	51	16	29.97
random		2	3253-3319	3320-3394	3394-3466	11	37	52	40.10
geometric	500	5	4-13	14-23	24-37	7	58	35	8.26
geometric		10	35-59	60-84	85-123	19	42	39	9.50
geometric		20	148-246	247-346	347-450	41	44	15	11.40
geometric		40	441-840	841-1240	1241-3400	47	20	33	14.50
geometric	1000	5	24-43	44-63	64-78	37	57	6	18.70
geometric		10	65-114	115-164	165-205	16	54	30	21.20
geometric		20	196-399	400-599	600-816	32	60	8	23.84
geometric		40	537-1099	1100-1599	1600-5829	28	49	23	28.56

* $T = 10$; $TF = 0.95$; $SF = 16$; $chPerc = 2\%$; $Cutoff = 10\%$.

[†]For random class, numbers represent density; for geometric class, numbers represent $n\pi d^{\dagger}$.

LOCALIZER. For a given benchmark, all ranges have the same width, and the lower bound of the first range is the best solution found. The upper bound of the last range, when

present, gives the worst solution ever returned. If absent, it means that LOCALIZER produced at least one solution that would not fit in the range.

Table V. Graph Partitioning: Comparison Results

Graph			Results*				
Class	Vertices	Density	L. Best	L. Time	J. Best	J. Time	Ratio
random	124	2	11	2.22	13	85.4	38.5
		4	55	2.4	63	82.2	34.3
		8	159	3.24	178	78.1	24.1
		16	481	4.05	449	104.8	25.9
random	250	1	20	4.68	29	190.6	40.7
		2	92	5.1	114	163.7	32.1
		4	324	6.5	357	186.8	28.7
		8	828	9.98	828	223.3	22.4
random	500	0.5	47	10.08	52	379.8	37.7
		1	219	10.76	219	308.9	28.7
		2	635	14.44	628	341.5	23.6
		4	1661	20.67	1744	432.9	20.9
random	1000	0.25	90	19.99	102	729.9	36.5
		0.5	439	22.62	451	661.2	29.2
		1	1326	29.97	1367	734.5	24.5
		2	3253	40.01	3389	853.7	21.3

* $T = 10$; $TF = 0.95$; $SF = 16$; $chPerc = 2\%$; $Cutoff = 10\%$.

Efficiency.

Table V compares LOCALIZER with the results reported in [4]. It is important to mention that [4] explicitly writes that their results are very hard to reproduce, because they chose the temperature of the annealing algorithm according to some preliminary observation of its behavior on each class of graphs. LOCALIZER, in contrast, is always run with the given parameters. In addition, the relevant results are only given for the random graphs in [4]. The times for LOCALIZER are given on a SUN Sparc Ultra-1 running Solaris 5.5.1 and the standard C++ compiler, while the results in [4] are given for a slow VAX-750. In general, the quality of the results produced by LOCALIZER is slightly better than the quality in [4]. Once again, the performance results indicate that LOCALIZER behaves well on these problems.

6. Conclusion

The main contribution of this article is to show that local search can be supported by modeling languages to shorten the development time of these algorithms substantially while preserving the efficiency of special-purpose algorithms. To validate this claim, we presented the preliminary design of the modeling language LOCALIZER. LOCALIZER statements are organized around the traditional concepts of local search and may exploit the special structure of the problem at hand. The main conceptual tool underlying LOCALIZER is the concept of invariant, which makes it possible to specify complex data structures de-

claratively. These data structures are maintained incrementally by LOCALIZER, automating one of the most tedious and error-prone parts of local search algorithms. Preliminary experimental results indicate that LOCALIZER can be implemented to run with an efficiency comparable to specific implementations.

Our current research focuses on building a large collection of applications in LOCALIZER to understand the strengths and limitations of the language, on implementing extensions to support tabu search^[31] at a higher level (e.g., automating many of the tedious aspects still needed in LOCALIZER), and on studying other frameworks such as dynamic k -opt,^[8] genetic algorithms, and the integration of constraint techniques. Longer-term research will explore how LOCALIZER can be turned into a programming language library to guarantee extensibility and wide applicability for expert users, while preserving the right level of abstraction.

Acknowledgments

A preliminary version of Research under article appeared in the proceedings of the International Conference on Constraint Programming (CP'97). This paper is dedicated to the memory of Paris C. Kanellakis, who kept on gently pressuring us to pursue this topic. Thanks to D. McAllester and B. Selman for many discussions on this research, to Costas Bush for implementing the graph-coloring algorithm in C, and to one of the reviewers for clarifying the relationship between LOCALIZER and traditional modeling languages. This research was partly supported by the Office of Naval Research under Grant N00014-94-1-1153, the National Science Foundation under Grant CCR-9357704, and a National Science Foundation National Young Investigator Award. A preliminary version of LOCALIZER is available by anonymous ftp (ftp.cs.brown.edu) or from the authors' web pages.

References

1. R. FOURER and J. BISSCHOP, 1996. New Constructs for the Description of Combinatorial Optimization Problems in Algebraic Modeling Languages. *Computational Optimization and Applications* 6, 83–116.
2. R. FOURER, D. GAY, and B. KERNIGHAN, 1993. *AMPL: A Modeling Language for Mathematical Programming*. Scientific Press, San Francisco, CA.
3. F. GLOVER, 1989. Tabu Search. *ORSA Journal on Computing* 1, 190–206.
4. D. JOHNSON, C. ARAGON, L. MCGEOCH, and C. SCHEVON, 1989. Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning. *Operations Research* 37, 865–893.
5. D. JOHNSON, C. ARAGON, L. MCGEOCH, and C. SCHEVON, 1991. Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring and Number Partitioning. *Operations Research* 39, 378–406.
6. S. KIRKPATRICK, C. GELATT, and M. VECCHI, 1983. Optimization by Simulated Annealing. *Science* 220, 671–680.

7. S. MINTON, M. JOHNSTON, and A. PHILIPS, 1990. Solving Large-Scale Constraint Satisfaction and Scheduling Problems using a Heuristic Repair Method, *AAAI-90*, 17–24.
8. C. PAPADIMITRIOU and K. STEIGLITZ, 1982. *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ.
9. B. SELMAN and H. KAUTZ, 1993. An Empirical Study of Greedy Local Search for Satisfiability Testing, *AAAI-93*, 46–51.
10. B. SELMAN, H. LEVESQUE, and D. MITCHELL, 1992. A New Method for Solving Hard Satisfiability Problems, *AAAI-92*, 440–446.
11. P. STUCKEY and V. TAM, 1996. Models for Using Stochastic Constraint Solvers in Constraint Logic Programming, *PLILP-96*, 423–437.
12. P. VAN HENTENRYCK, L. MICHEL, and Y. DEVILLE, 1997. *Numerica: a Modeling Language for Global Optimization*, MIT Press, Cambridge, MA.