Localizer: A Modeling Language for Local Search

Laurent Michel and Pascal Van Hentenryck

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-97-03
April 1997

LOCALIZER
A Modeling Language for Local Search

Laurent Michel and Pascal Van Hentenryck
Brown University, Box 1910
Providence, RI 02912 (USA)
Email: {ldm,pvh}@cs.brown.edu

Abstract

Local search is a traditional technique to solve combinatorial search problems which 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.

1 Introduction

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 [1] and NUMERICA [10] to constraint programming languages such CHIP, ILOG SOLVER, CLP($\Re$), PROLOG-IV, and OZ to name only a few. In contrast, little attention has been devoted to the support of local search (LS), despite the increasing interest in these algorithms in recent years. (Note however there are various efforts to integrate local search in CLP languages, e.g., [9]). The design of LS algorithms is not an easy task however. The same problem can be modeled in many different ways (see for instance [3]), 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 lan-

guage which 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 [6], simulated annealing [4], tabu search [2], and GSAT [8]), 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.

The main contribution of the paper is to show that local search algorithms can be supported by very high-level modeling languages which shorten their development time substantially while preserving most of the efficiency of special-purpose implementations. The paper 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 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 an informal description of some of the features of LOCALIZER since space limitations do not allow to cover the full language in detail. Section 2 is a brief tour of LOCALIZER statements. Sections 3 and 4 illustrate LOCALIZER on two problems: satisfiability (decision problem) and graph coloring (optimization problem). Section 5 contains the experimental results and Section 6 concludes the paper.

2 A Brief 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 *MaxTries* and while a global condition is satisfied). Each local search consists of a number of iterations (up to *MaxIterations* 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 *MaxIterations*, the algorithm restarts a new 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,

```
procedure LOCALIZER
begin
 1  s := startState();
 2  for i := 1  to MaxTries while Gcondition  do
 3      for j := 1  to MaxIterations while Lcondition  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

```
 4          if value(s) < f* then
 4.1            f* := value(s);
 4.2            best := s;
```

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

```
⟨ Model ⟩ ::= [⟨DataType⟩]
              ⟨Data⟩
              ⟨Variable⟩
              [ ⟨Invariant⟩]
              ⟨Operator⟩
              ⟨Satisfaction⟩
              [ ⟨Objective Function⟩]
              ⟨Neighborhood⟩
              ⟨Start⟩
              [ ⟨Restart⟩]
              [ ⟨Parameter⟩]
              [ ⟨Global Condition⟩]
              [ ⟨Local Condition⟩]
              [ ⟨Init⟩]
```

Figure 2: The Structure of LOCALIZER Statements

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`, `Data`, 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. *MaxIterations*) are given in the `Parameter` section, and the global and local conditions are given in Sections `Global Condition` and `Local Condition`. As mentioned previously, the most original aspects of LOCALIZER are in the specifications of the neighborhood and the acceptance criterion.

Neighborhood The neighborhood in LOCALIZER is defined through basic instructions of the form

select [**random** | **best**]
  $op(x_1, \ldots, x_n)$
where
  x_1 **in** S_1;
  ...
  x_n **in** S_n
accept when
  ⟨ Acceptance ⟩

where *op* is an operator or a sequence of instructions using traditional programming language constructs. In its simplest form, 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

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

selects one of its elements, and checks if it satifies the acceptance criterion. When the keyword **best** is present, the instruction selects an element optimizing the value of the objective function and tests it against the acceptance criterion. When the keyword **random** is present, the instruction selects an element of the neighborhood in a random way (which is important in many local search algorithms) and tests it agains the acceptance criterion. The sets in the **select** statements are specified using the syntax:

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

For instance, an instruction of the form

select best $a[i] := !a[i]$
where i **in** $1..n$
accept when ...;

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

```
select q[i] := v
where
    i in Conflicts;
    v in 1..n minimizing
        sizeof({ j in 1..n | q[j] = v or q[j] = v + i − j or q[j] = v + j − i});
accept when ...;
```

is an example of min-conflict heuristics of [5]. The GSAT statement simply states 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 which 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** after **select**.

Select instructions can also be composed to consider various neighborhoods using the **try** instruction:

```
try
    [Probability:] ⟨ Select Statement ⟩;
    ...
    [Probability:] ⟨ Select Statement ⟩;
end
```

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

```
try
    0.1:
        select a[i] := !a[i]
        where i in OccurInUnsatClause
        accept when ...;
    default:
        select best a[i] := !a[i]
        where i in 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 the LOCALIZER simply goes to the next iteration if the selected neighborhood is empty, since other neighborhoods may be non-empty.

Invariants The specification of the set expressions $S_1, \ldots, S_n$ in the select instructions is probably the most interesting part of the neighborhood definitions.

These sets are 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 $v = exp$ and LOCALIZER guarantees that, at any time during the computation, the value of variable v is the value of the expression exp. Typical examples of invariants are

$v =$ **sum**(i **in** S) $a[i]$
$C = \{\ i$ **in** $S \mid a[i] = 0\ \}$
$D[\ i$ **in** $I\ ] = \{\ j$ **in** $S \mid a[j] = i\ \}$.

The first invariant specifies that v is the summation of the $a[i]$ ($i \in S$), the second invariant specifies that C 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 using 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.

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, whenever a value $a[k]$ is changed, v, C, and $D[j]$ are updated in constant time in our current implementation.[1] As a consequence, invariants make it possible to specify **what** needs to be maintained incrementally without considering **how** to do so.

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

⟨ Acceptance ⟩ ::= ⟨ Criterion ⟩ [→ ⟨ Statement ⟩]
⟨ Criterion ⟩ ::= **always** |
improvement |
noDecrease |
⟨ Expr ⟩ |
⟨ Criterion ⟩ **and** ⟨ Criterion ⟩ |
⟨ Criterion ⟩ **or** ⟨ Criterion ⟩ |
not ⟨ Criterion ⟩

The main part of the acceptance statement is the acceptance criterion. The optional statement is useful to implement some termination condition and it is discussed later on. Keyword `always` accepts all moves, `improvement` accepts a move only if it improves the value of the objective function, and `noDecrease` accepts a move only if it does not degrade the value of the objective function. The acceptance criteria 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

[1] Of course, other constructs cannot be updated in constant time. A complete description of the algorithms cannot be included for space reasons.

```
select
   ...
accept when improvement;
```

while a tabu-search strategy is specified using a statement of the form

```
select
   ...
where
   o in S
accept when
   o notin Tabu
```

Acceptance statements can also be combined using an instruction of the form

```
try
   [ Probability: ] ⟨ Acceptance ⟩;
   ...
   [ Probability: ] ⟨ Acceptance ⟩;
end
```

which tries each acceptance criterion in sequence until one has succeeded or all have failed. 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. For instance, a typical simulated annealing procedure uses the acceptance criterion

```
try
   improvement;
   e^(-delta/t): always;
end
```

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 $e^{-\mathbf{delta}/t}$.

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 which terminates when the objective function has been stable for a number of iterations, one would write:

```
Neighborhood:
   select
   ...
   accept when
      try
         improvement → nbStableIter := 0;
         noDecrease → nbStableIter++;
      end;
Local Condition:
   nbStableIter ≤ maxStableIter;
```

3 Satisfiability

We now illustrate LOCALIZER on a number of applications. Our first application is satisfiability (SAT) and we illustrate how GSAT [8] can be expressed in LOCALIZER. GSAT illustrates many aspects of LOCALIZER as well as several interesting modeling issues.

3.1 The Local Search

Problems in GSAT 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 asssignment 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 which, when its value is inverted, produces a state with the largest number of clauses satisfied. Note that GSAT accepts only moves which do not decrease the number of clauses satisfied.

3.2 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: its set of positive atoms `p` and its set of 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.

Neighborhood The neighborhood uses two invariants: the number of true literals for clause i, denoted by $nbtl[i]$, and the number of clauses satisfied, denoted by *nbClauseSat*. $nbtl[i]$ is computed by counting the number of positive atoms and the negation of the negative atoms. *nbClauseSat* simply sums the $nbtl[i]$ for all 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 which 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 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.

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

Figure 3: A Simple Model of GSAT

3.3 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 amenable easily to such optimizations.

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

Figure 4: A More Incremental Model of GSAT

Instance Representation The representation is the same as in the simple model. However, the `Data` 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, since this information is already expressed in the invariants. Note that some invariants in this model involve sets, conditional expressions, and aggregation operators which 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.

3.4 Adding Weights

Reference [7] 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.,

$g01[\ i \text{ in } 1..n\ ] = \mathbf{sum}(j \mathbf{\ in\ } po[i])\ w[j]{\times}(a[j] = 0)$ - $\mathbf{sum}(j \mathbf{\ in\ } no[i])\ w[j]{\times}(a[j] = 1)$;
$g10[\ i \mathbf{\ in\ } 1..n\ ] = \mathbf{sum}(j \mathbf{\ in\ } no[i])\ w[j]{\times}(a[j] = 0)$ - $\mathbf{sum}(j \mathbf{\ in\ } po[i])\ w[j]{\times}(a[j] = 1)$;

and in updating the weights after each local search by adding a restarting section

Restart:
  forall(i in $1..m$)
    $w[i] := w[i] + (a[i] = 0)$;

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

4 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 two adjacent vertices have a different color. It shows how a simulated annealing algorithm proposed in [3] can be expressed in LOCALIZER. Of particular interest is once again the close similarity between the problem description and the model. Note 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.

The Local Search For a graph with n vertices, the algorithm considers n colors which 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 since 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.

The Model The model is depicted in Figure 5 and it closely follows the above description. The state is given by the variables $v[i]$ which represent the colors of vertices and by the temperature t. The invariants describe the set C_i, B_i, and the objective function. The unused color is obtained by taking the smallest unused color. The set of candidate colors are thus all the "used" colors together with the selected unused color. The total number of bad edges is also 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) and the temperature is updated in the restarting section. Once again, the design of the model is essentially a direct formalization of the informal statement of the algorithm and the distance between the model and the informal statement is small. The main component of the model is the specification of the data structures which can be done declaratively, without specifying the implementation details.

5 Experimental Results

This section describes some preliminary results on the implementation of LOCALIZER (about 10,000 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

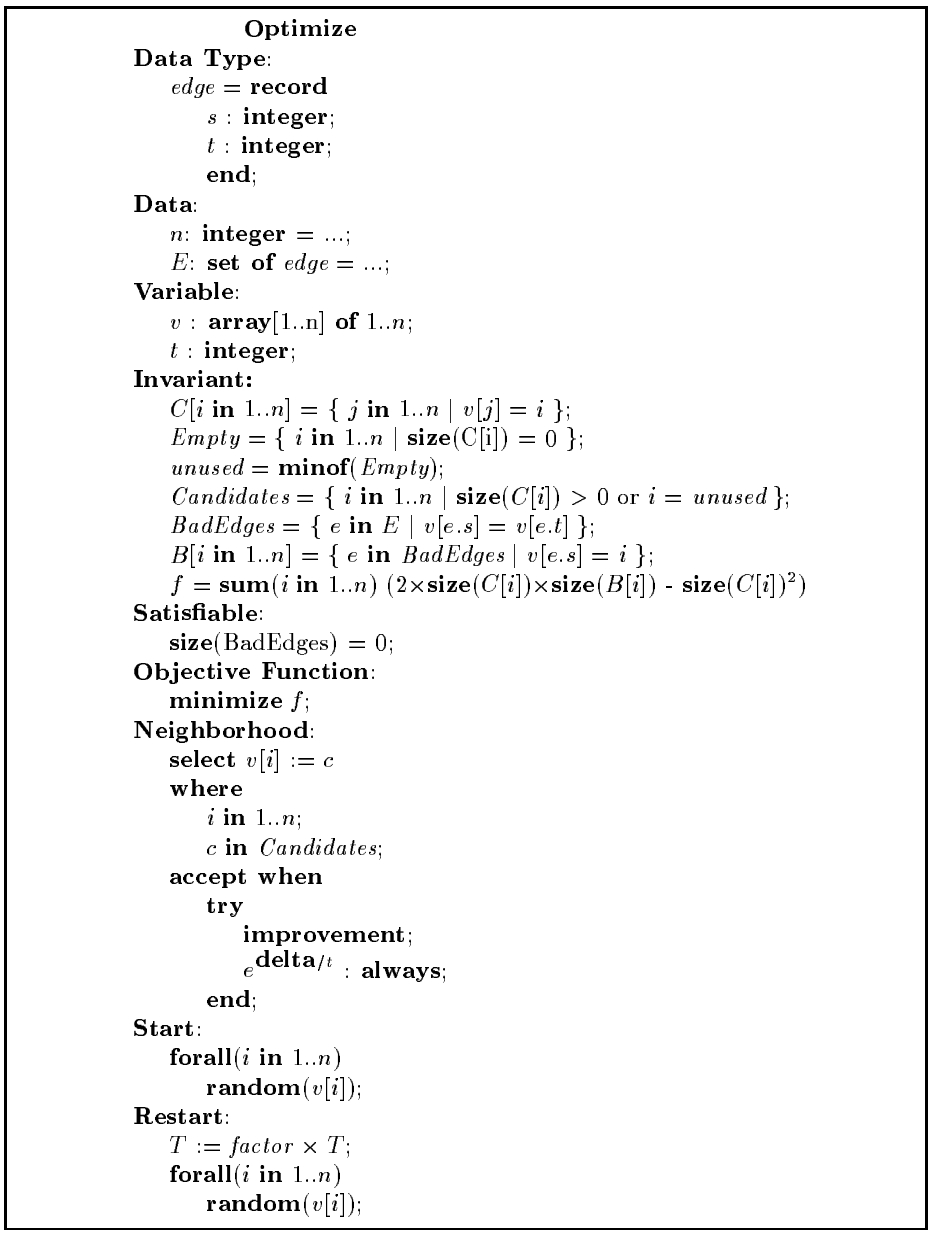

Figure 5: A Graph Coloring Model

demonstrate practicability, we compare LOCALIZER with GSAT, which is generally recognized as a fast and very well implemented system. The experimental results were carried out as specified in [8]. Table 1 gives the number of variables (V), the number of clauses (C), and `MaxIterations` (I) for each class of benchmarks as well as the CPU times in seconds of LOCALIZER (L), the CPU times in seconds of GSAT (G) as reported in [8], and the ratio L/G. The times of GSAT are given on a SGI Challenge with a 70 MHz MIPS R4400 processor. The times of LOCALIZER were obtained on a SUN SPARC-10 40MHz and scaled 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.

	V	C	I	L	G	R
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
Av.						3.38

Table 1: Experimental Results

As can be seen, GSAT is 3.38 times faster than LOCALIZER on the benchmarks and the two systems scale in about the same way. The gap between the two systems is about one machine generation (i.e., on modern workstations, LOCALIZER runs as efficiently as GSAT on machines of three years ago), which is really acceptable given the preliminary nature of our (unoptimized) implementation.

6 Conclusion

The main contribution of this paper 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 make it possible to specify complex data structures declaratively. 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, to study how frameworks such as Guided Local Search [11] and dynamic k-opt [6] can be supported, and to identify which extensions are needed to turn our initial design into a practical tool. We are also contemplating extensions of the language to support genetic algorithms (e.g., maintaining multiple configurations) and to integrate consistency techniques to enhance the expressivity of invariants.

Acknowledgments Thanks to D. McAllester and B. Selman for many discussions on this research. This research was supported in part by ONR under grant ONR Grant N00014-94-1-1153 and an NSF National Young Investigator Award.

References

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

[2] F. Glover. Tabu Search. *Orsa Journal of Computing*, 1:190–206, 1989.

[3] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon. Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring and Number Partitioning. *Operations Research*, 39(3):378–406, 1991.

[4] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, 1983.

[5] S. Minton, M.D. Johnston, and A.B. Philips. Solving Large-Scale Constraint Satisfaction and Scheduling Problems using a Heuristic Repair Method. In *AAAI-90*, August 1990.

[6] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, NJ, 1982.

[7] B. Selman and H. Kautz. An Empirical Study of Greedy Local Search for Satisfiability Testing. In *AAAI-93*, pages 46–51, 1993.

[8] B. Selman, H. Levesque, and D. Mitchell. A New Method for Solving Hard Satisfiability Problems. In *AAAI-92*, pages 440–446, 1992.

[9] P. Stuckey and V. Tam. Models for Using Stochastic Constraint Solvers in Constraint Logic Programming. In *PLILP-96*, Aachen, August 1996.

[10] P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica: a Modeling Language for Global Optimization.* The MIT Press, Cambridge, Mass., 1997.

[11] C. Voudouris and E. Tsang. Partial Constraint Satisfaction Problems and Guided Local Search. In *PACT-96*, London, April 1996.