Localizer++: An Open Library for Local Search

Laurent Michel and Pascal Van Hentenryck

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-01-02
January 2001

`Localizer++`: An Open Library for Local Search

Laurent Michel and Pascal Van Hentenryck
Brown University
Box 1910
Providence, RI 02912
Email: `pvh@cs.brown.edu`

Abstract

Local search is one of the fundamental approaches to tackle large combinatorial optimization problems. Yet relatively little support is available to facilitate the design and implementation of local search algorithms. This paper introduces `Localizer++`, an extensible object-oriented library for local search. `Localizer++` supports both declarative abstractions to describe the neighborhood and high-level search constructs to specify local moves and meta-heuristics. It also supports a variety of features typically found only in modeling languages and its extensibility allows for an easy integration of new, user-defined, abstractions. Of particular interest is the conciseness and readability of `Localizer++` statements and the efficiency of the `Localizer++` implementation.

1 Introduction

Combinatorial optimization problems are ubiquitous in many practical applications, including scheduling, resource allocation, planning, and configuration problems. These problems are computationally difficult (i.e., they are NP-hard) and require considerable expertise in optimization, software engineering, and the application domain.

The last two decades have witnessed significant progress in optimization tools in order to simplify the design and implementation of combinatorial optimization problems. Their goal is to decrease development time substantially while preserving most of the efficiency of specialized programs. Much of this effort was devoted to mathematical and constraint programming systems for which there are now advanced modeling languages (e.g., [10, 3, 38]), constraint programming languages (e.g., [6, 4, 7, 33, 36, 20]) and libraries (e.g., [14, 26]). Less attention was paid to local search, although it has been shown particularly effective for many classes of applications. However, the implementation of local search algorithms may present interesting challenges in order to obtain high-quality solutions in reasonable time. Indeed, these algorithms often require maintaining complex data structures incrementally to obtain good performance (e.g., [18, 21, 23]). They may also require an important programming effort to implement the meta-heuristics needed to obtain good-quality solutions. In addition, despite considerable research (e.g., [1]), the design of local search algorithms remains an art and considerable experimentation is often necessary before converging to a satisfactory solution. As a consequence, there is a strong need for expressive and efficient optimization tools for local search.

This lack of support is increasingly recognized and recent work has started to address it. Most of the work in this area (e.g., [32, 5, 9, 2]) has focused on providing frameworks for local search. These frameworks offer a general structure for local search algorithms and factorize generic aspects of meta-heuristics. In addition, a number of domain-specific languages were also proposed to simplify the specification of the neighborhood and of local moves. `Localizer` [23, 24, 25], the first modeling language for local search, introduced the concept of *invariants* to specify incremental data structures and supported a number of high-level abstractions to describe local move and meta-heuristics. `Localizer` demonstrated that a modeling language may greatly reduce the development time of local search applications, while inducing a reasonable overhead in efficiency. Unfortunately, `Localizer` is a closed system which is not easy to extend or integrate in a larger application. `Salsa` [20] is another interesting domain-specific language offering high-level abstractions for specifying global and local search procedures. Some of these abstractions were closely related to those of `OPL` [38] and `Localizer` [24] and were shown to describe search procedures concisely and naturally. Finally, a small constraint language, whose implementation is based on local search, is presented in [11]. Unfortunately, this system is a black-box where users have no control over the search procedure and have limited ways of expressing their constraints.

The purpose of this paper is to show that local search algorithms can be supported elegantly in an object-oriented programming library. In particular, it presents the design of `Localizer++`, an extensible `C++` library for local search which supports both declarative abstractions to describe the neighborhood and high-level search constructs to specify local moves and meta-heuristics concisely. In addition, since it is based on the `OPL++` modeling layer for optimization libraries [26], `Localizer++` supports a variety of features typically found only in modeling languages, such as high-level set, algebraic, logical, and array operators, multi-dimensional arrays, and sparse data structures. The extensibility of `Localizer++` also enables users to implement their own invariants and search constructs easily.

`Localizer++` offers a number of significant benefits. First, it automates, inside a mainstream object-oriented programming language, some of the most tedious and error-prone aspects of many local search algorithms. As consequence, it leads to higher-level programs that are easier to implement and to maintain. Second, `Localizer++` programs, being implemented directly in a traditional language, are easy to integrate within a larger application. Third, its extensibility eases the implementation of vertical extensions for specific domains. Finally, `Localizer++` may greatly simplify the implementation of modeling languages for local search as well as their integration inside larger applications, since the modeling language and the library may now share the same data and control structures.

The rest of this paper illustrates `Localizer++` through a number of traditional local search applications, starting with simple examples such as the queens problem and concluding with the more realistic graph-partioning problem. It also shows how to extend `Localizer++` with new invariants and search constructs. The main focus of the paper is to report preliminary evidence that the approach taken in `Localizer++` may greatly simplify the implementation of many local search algorithms, while retaining excellent performance. Section 2 presents a high-level overview of `Localizer++`. The next three sections describe three applications of `Localizer++`. Section 3 presents a local search algorithm solving the n-queens in linear space and quadratic time. The algorithm is presented in stepwise refinements to culminate in a constraint-based formulation which solves very large instances of this problem (e.g., $n \geq 250,000$). Section 4 presents a tabu search algorithm for the magic square problem, an application which is extremely difficult for mathematical and constraint programming systems. It describes a short algorithm solving large instances (over 2500 variables). Section 5 describes an algorithm for graph-partitioning inspired by the linear heuristic of [8] and the very large scale neighborhood technique of Kernighan and Lin [17]. Section 6 shows how to add new invariants and search procedures and Section 7 concludes the paper. It is important to stress that the purpose of these applications is to illustrate `Localizer++`, not to present the best possible algorithms for these problems. For instance, it is simple to express, in `Localizer++`, the simulated annealing algorithm for graph-partitioning presented in [15]. The resulting `Localizer++` program runs with an efficiency comparable to the specific programs (as we already showed in [25]) but it does not illustrate as many features of `Localizer++` as the algorithm presented here.

2 A Brief Overview of `Localizer++`

The structure of `Localizer++` is similar in spirit to that of constraint programming languages. It is a two-level architecture composed of a declarative and a search component. At a very high level, the declarative component can be thought of as specifying the neighborhood, while the search component indicates how to explore it. `Localizer++` is also built on top of the `OPL++` modeling layer [26]. As a consequence, it inherits many modeling features typically found in modeling languages such as multidimensional arrays, sparse data structures, and high-level set, algebraic, and array notations. Finally, `Localizer++` is an extensible library which lets users extend the declarative and the search components.

The Declarative Component The declarative component is the core of the architecture. It is motivated by the recognition that many local search algorithms maintain complex data structures incrementally in order to achieve a reasonable efficiency (e.g., [16, 18]). These data structures describe the current node of the search space as well as its neighborhood. Since two neighboring nodes generally have many commonalities, incremental algorithms and data structures may produce orders of magnitude improvement in efficiency compared to naive implementations. The goal of the declarative component is to simplify this tedious and error-prone aspect of local search. The main abstraction of the declarative component is the concept of *invariant* [23, 24]. Invariants specify, in a declarative way, the data structures to maintain without defining how to update them incrementally. `Localizer++` features a wide range of invariants which include arithmetic, array, and set invariants. For instance, the arithmetic invariant

```
sum << OplSum(i,Size,a[i]))
```

specifies that `sum` is the summation of all elements in array `a` and the set invariant

```
BestMoves << OplSetOf(m,Moves,gain[m] == bestGain)
```

specifies that `BestMoves` is the set of elements in set `Moves` whose gains are equal to `bestGain`. `Localizer++` applies incremental algorithms to maintain these invariants under changes to their variables. These algorithms are, in general, optimal in the size of the input and output changes, a model of computational complexity proposed in [30, 31] to analyze incremental algorithms. For instance, when an element of `a` is changed in the above invariant, `Localizer++` updates variable `sum` in constant time. Similarly, the set variable `BestMoves` is updated in constant time when elements are added or deleted from set `Moves` or when an element in array `gain` is updated. As a consequence, `Localizer++` lets users focus on defining an appropriate neighborhood and relieves them from specifying how it should be maintained.

Once invariants are available as first-class objects, it becomes natural to support high-level constraints in `Localizer++`. These constraints capture interesting substructures that arise frequently in practice and are amenable to efficient implementation. They play a similar role to global constraints in constraint programming. Of course, constraints in local search and in constraint programming often have fundamentally different functionalities.[1] Instead of pruning the search space, constraints in local search maintain, in an incremental manner, information such as their satisfiability status, their degree of violation, as well as how this information is affected by local moves. They are generally implemented in terms of a collection of invariants. `Localizer++` supports a variety of constraints to simplify the implementation of local search algorithms. For instance, the constraint

```
OplLSAlldifferent alldiff(queen)
```

specifies that all variables in array `queen` must be given a different value. The satisfiability status and the degree of violation of this constraint as well as other useful information to guide the local search can be obtained by invoking methods on `alldiff`. For instance, the expression

```
alldiff.isTrue()
```

return true (1) if the constraint is satisfied in the current node and false (0) otherwise, while the expression

```
alldiff.getNumberOfViolations()
```

represents how many constraints are violated by the current node. Note that these expressions are maintained incrementally during the execution. The applications presented in this paper illustrate a variety of invariants and some high-level constraints since these are more problem-specific. Section 6.1 also shows how to extend the library with new invariants.

The Search Component The search component operates around the declarative component and is procedural in nature. Its main purpose is to choose a neighbor of the current node and to perform the move. Once the move is applied, `Localizer++` updates the invariants automatically. Observe that, although the declarative and search components are physically separated in the `Localizer++` code, they are interleaved during the actual computation. Substantial research in combinatorial optimization was devoted to designing good strategies to select the next move. These meta-heuristics, which include the variable-depth procedure of Kernighan & Lin, simulated

[1] Very large scale neighborhood search based on constraint programming (e.g., [35]) somewhat blurs this distinction. However, invariants seem also to be natural abstractions to implement the propagation rules needed in these algorithms.

annealing [19] and tabu search [12, 13] to name only three, are often critical in obtaining good quality solutions but they may also induce significant programming effort and experimentation. The search component of `Localizer++` aims at providing high-level abstractions to specify local moves, to simplify the implementation of these meta-heuristics, and to encourage experimentation. It features multidimensional *selection operators* to choose potential moves, *simulation operators* to evaluate the consequence of a given move, as well as *counters* which simplify the bookkeeping typically involved in maintaining forbidden moves in meta-heuristics. For instance, a search procedure may feature a *selection operator*

```
OplSelectMax(e,Set,violations[e])
```

to select an element `e` of `Set` maximizing `violations[e]`. Note that selection operators in `Localizer++` are stochastic in nature: ties are broken using a uniform distribution, since this is often critical in practical applications. They are also multidimensional, i.e., the selection may involve several parameters, as illustrated later in the paper. A search procedure may also contain a *simulation operator* of the form

```
OplLookahead(Obj,Move)
```

to evaluate the effect of a move `Move` on the expression `Obj`. Once again, the applications in this paper illustrate a variety of search abstractions of `Localizer++` and Section 6.2 also shows how to extend the library with new ones.

3 The Queens Problem

This section describes local search algorithms for the n-queens problem, i.e., the problem of placing n queens on a chessboard of size $n \times n$ so that no two queens attack each other. The simplicity of n-queens problem makes it an excellent vehicle to introduce many aspects of `Localizer++`. In addition, it allows us to demonstrate the time and memory efficiency of the implementation since the `Localizer++` statement runs in linear space and quadratic time and solves very large instances of this problem. This rest of this section is organized as follows. Section 3.1 presents a high-level modeling of the problem, Section 3.2 describes a first `Localizer++` statement which is quadratic in space and cubic in time, Section 3.3 shows a linear-space and quadratic-time implementation, and Section 3.4 gives a constraint-based implementation with the same complexities. This stepwise presentation should give readers precise insights on how invariants are instrumental in defining high-level constraints.

3.1 High-Level Modeling

The local search algorithm presented in this section is based on the min-conflict heuristic [27]. The algorithm starts with an initial random configuration. Then, at each iteration, it chooses the queen violating the largest number of constraints and moves it to a position minimizing its violations. This step is iterated until a solution is found or the algorithm reaches a limit on the number of iterations. Since a queen must be placed on every column, the algorithm uses an array `queen` of variables and `queen[i]` denotes the row of the queen placed on column `i`.

The declarative component of the `Localizer++` statement maintains the number of conflicts for all possible positions of the queens. Since the problem constraints are given by

```
queen[i] <> queen[j]
queen[i] + i <> queen[j] + j
queen[i] - i <> queen[j] - j
```

for all $1 \leq \texttt{i} < \texttt{j} \leq \texttt{n}$, the algorithm uses three arrays of variables: `queen[i]` denotes the row of the queen placed on column `i` as mentioned earlier, `queenPos[i]` denotes `queen[i] + i` and `queenNeg[i]` denotes `queen[i] - i`. For each of these arrays, the algorithm maintains the number of occurrences of each possible value. Obviously, a conflict arises as soon as a value occurs more than once, since such a configuration violates the above constraints. More precisely, an array `r` contains the number of occurrences of each value `v` in array `queen`. The value `r[v]`, which denotes how many queens are assigned to row `v`, can be defined as

$$\texttt{r}[\texttt{v}] = \sum_{\texttt{q}=1}^{\texttt{n}} (\texttt{queen}[\texttt{q}] = \texttt{v})$$

where the expression `queen[q] = v` evaluates to 1 if `queen[q]` is equal to `v` and 0 otherwise. Similar definitions can be given for the second and third types of constraints; they use the `queenPos` and `queenNeg` arrays and store the violations in array `ld` and `ud` respectively.

The search component of the `Localizer++` statements simply selects a queen with the largest number of conflicts and moves it to a position minimizing its number of violations. The conflict of a queen `q` is given by the formula

```
r[queen[q]] + ld[queenPos[q]] + ud[queenNeg[q]]
```

and its new position is a row `v` minimizing

```
r[v] + ld[v+q] + ud[v-q].
```

3.2 A First `Localizer++` Statement

We are now in position to describe a first `Localizer++` statement for this local search algorithm. The statement, which is depicted in Figure 1, follows closely the high-level description just presented and serves as a good vehicle to introduce many fundamental concepts of `Localizer++`. The first instructions

```
OplLSManager mgr;
OplIntRange Size(mgr,1,n), SizeP(mgr,2,2*n), SizeN(mgr,1-n,n+1);
OplIntParameter q(mgr),v(mgr);
```

```
void queens(int n)
{
   OplLSManager mgr;
   OplIntRange Size(mgr,1,n), SizeP(mgr,2,2*n), SizeN(mgr,1-n,n+1);
   OplIntParameter q(mgr),v(mgr);

   OplIntIncArray queen(mgr,Size,"queen");
   OplUniformDistribution d(mgr,Size);
   for (int k = 1; k <= n ; k++)
      queen[k] = d.get();
   OplIntIncArray queenPos(mgr,Size,"queenPos");
   OplIntIncArray queenNeg(mgr,Size,"queenNeg");
   OplIntIncArray r(mgr,Size,"r"), ld(mgr,SizeP,"ld"), ud(mgr,SizeN,"ud");
   OplIntInc success(mgr);

   mgr.post(OplForall(q,Size,queenPos[q] << queen[q] + q));
   mgr.post(OplForall(q,Size,queenNeg[q] << queen[q] - q));
   mgr.post(OplForall(v,Size,r[v] << OplSum(q,Size,queen[q] == v)));
   mgr.post(OplForall(v,SizeP,ld[v] << OplSum(q,Size,queenPos[q] == v)));
   mgr.post(OplForall(v,SizeN,ud[v] << OplSum(q,Size,queenNeg[q] == v)));
   mgr.post(success <<
      (OplAnd(v,Size,r[v]<=1) && OplAnd(v,SizeP,ld[v]<=1) && OplAnd(v,SizeN,ud[v]<=1)));

   OplLSAtom move =
      OplAnd(OplSelectMax(q,Size,r[queen[q]]+ld[queenPos[q]]+ud[queenNeg[q]]),
             OplSelectMin(v,Size,r[v]+ld[v+q]+ud[v-q]),
             queen[q] <<= v);

   OplLSSearch search = mgr.newSearch();
   for(int it = 0; it <= 20*n; it++) {
      if (success.getValue()) break;
      search.commit(move);
   }
}
```

Figure 1: The Queens Problem in `Localizer++` (First Version).

declare a local search manager, three ranges, and two parameters. The local search manager is the entry point of `Localizer++`. It manages the declarative and search components. Parameters, which are inherited from `OPL++`, are fundamental tools to express invariants and search procedures as will become clear shortly. The next instruction

```
OplIntIncArray queen(mgr,Size,"queen");
```

declares the array `queen` of *incremental integer variables* to represent the solutions. *Incremental variables* are one of the fundamental concepts of `Localizer++` and they can be of various types, including integer, tuple, set of integers, or set of tuples. From a user standpoint, incremental variables behave very much like ordinary variables. From an implementation standpoint, they are of course the cornerstone of the declarative and search components. The instructions

```
OplUniformDistribution d(mgr,Size);
for (int k = 1; k <= n ; k++)
   queen[k] = d.get();
```

declares a uniform distribution and uses it to initialize the array `queen`. As can be seen, probability distributions are first-class objects in `Localizer++`. The next instructions

```
OplIntIncArray queenPos(mgr,Size,"queenPos");
OplIntIncArray queenNeg(mgr,Size,"queenNeg");
OplIntIncArray r(mgr,Size,"r"), ld(mgr,SizeP,"ld"), ud(mgr,SizeN,"ud");
OplIntInc success(mgr);
```

declare the remaining arrays of incremental variables, which help in keeping track of constraint violations as well as the variable `success` whose value is 1 if the configuration is a solution and is 0 otherwise.

The Declarative Component The next instructions are the declarative component: they express the invariants and post them to the local search manager which maintains them incrementally under changes to the `queen` variables. Observe that they are almost direct encodings of the formulas given in the high-level description and let us review some of them. The invariant

```
OplForall(q,Size,queenPos[q] << queen[q] + q)
```

defines the values in array `queenPos` in terms of the variables of `queen`. The invariant

```
OplForall(v,Size,r[v] << OplSum(q,Size,queen[q] == v))
```

defines `r[v]` as the number of queens placed on row `v`. It is a direct encoding of the formula given previously. The invariant

```
success << (OplAnd(v,Size,r[v]<=1) && OplAnd(v,SizeP,ld[v]<=1) && OplAnd(v,SizeN,ud[v]<=1))
```

defines `success` to be true (1) whenever no value occurs more than once in each of the arrays `r`, `ld`, and `ud`.

The Search Component The rest of the statement is the search component. The instruction

```
OplLSAtom move =
   OplAnd(OplSelectMax(q,Size,r[queen[q]]+ld[queenPos[q]]+ud[queenNeg[q]]),
          OplSelectMin(v,Size,r[v]+ld[v+q]+ud[v-q]),
          queen[q] <<= v);
```

declares the move of the local search. It consists of selecting a queen `q` with the largest number of conflicts, of selecting a new value `v` which induces the smallest number of conflicts for queen `q`, and of assigning the value `v` to queen `q`. It is important to mention that operators `OplSelectMax` and `OplSelectMin` break ties randomly, since this is an important feature for many applications. Observe also the conciseness of the move specification. The remaining instructions

```
OplLSSearch search = mgr.newSearch();
for(int it = 0; it <= 20*n; it++) {
   if (success.getValue()) break;
   search.commit(move);
}
```

specify the meta-heuristic. The first instruction defines a local search `search`. The remaining instructions apply a series of moves. The computation terminates when a solution is found or when the limit on the number of iterations is reached. The instruction

```
search.commit(move);
```

performs a move. The move is committed, which means that `Localizer++` will not be able to come back to a previous configuration. Note that Section 6.2 will illustrate other operations on local moves.

3.3 A Linear-Space and Quadratic-Time `Localizer++` Implementation

The `Localizer++` statement just presented behaves reasonably well in practice. Unfortunately, it is quadratic in space and cubic in time.[2] The culprits are the invariants of the form

```
OplForall(v,Size,r[v] << OplSum(q,Size,queen[q] == v))
```

which iterate over all queens and over all their possible values. The key idea to obtain a linear implementation is to cluster these invariants together to obtain a *global* invariant

```
OplCardIndicesOf(queen,r)
```

whose semantics is exactly the same as the above invariants. The global invariant `OplCardIndicesOf` arises in many applications (e.g., [24]) and can easily be implemented in linear time and space as shown later in this paper. The resulting `Localizer++` statement is shown in Figure 2. It solves the queens problem in linear space and quadratic time. In general, it requires a linear number of iterations to converge to a solution and each iteration takes linear time, since it must select a queen with the largest number of conflicts and a value minimizing the conflicts of the selected queen.

[2] As shown in [22], invariants can be analyzed systematically for space and time requirements.

```
void queens(int n)
{
   OplLSManager mgr;
   OplIntRange Size(mgr,1,n), SizeP(mgr,2,2*n), SizeN(mgr,1-n,n+1);
   OplIntParameter q(mgr),v(mgr);

   OplIntIncArray queen(mgr,Size,"queen");
   OplUniformDistribution d(mgr,Size);
   for (int k = 1; k <= n ; k++)
      queen[k] = d.get();
   OplIntIncArray queenPos(mgr,Size,"queenPos");
   OplIntIncArray queenNeg(mgr,Size,"queenNeg");
   OplIntIncArray r(mgr,Size,"r"), ld(mgr,SizeP,"ld"), ud(mgr,SizeN,"ud");
   OplIntInc success(mgr);

   mgr.post(OplForall(q,Size,queenPos[q] << queen[q] + q));
   mgr.post(OplForall(q,Size,queenNeg[q] << queen[q] - q));
   mgr.post(OplCardIndicesOf(queen,r));
   mgr.post(OplCardIndicesOf(queenPos,ld));
   mgr.post(OplCardIndicesOf(queenNeg,ud));
   mgr.post(success <<
      (OplAnd(v,Size,r[v]<=1) && OplAnd(v,SizeP,ld[v]<=1) && OplAnd(v,SizeN,ud[v]<=1)));

   OplLSAtom move =
      OplAnd(OplSelectMax(q,Size,r[queen[q]]+ld[queenPos[q]]+ud[queenNeg[q]]),
             OplSelectMin(v,Size,r[v]+ld[v+q]+ud[v-q]),
             queen[q] <<= v);

   OplLSSearch search = mgr.newSearch();
   for(int it = 0; it <= 20*n; it++) {
      if (success.getValue()) break;
      search.commit(move);
   }
}
```

Figure 2: The Queens Problem in `Localizer++` (A Linear-Space and Quadratic-Time Version).

Bench	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
CBS	0.00	0.01	0.01	0.02	0.07	0.15	0.67	2.63	10.96	47.82	185.91	736.22	2977	16726
ICBS	0.01	0.02	0.01	0.02	0.06	0.16	0.43	1.39	5.04	18.58	68.58	260.77	1001	10096

Table 1: Computation Times for Queens Problem in Seconds.

3.4 A Constraint-Based `Localizer++` Statement

A solution to the queens problem can be seen as an assignment of values to the variables in `queen` such that all variables in array `queen`, in array `queenPos`, and in array `queenNeg` are given distinct values. Alldifferent constraints, and more generally cardinality constraints, arise naturally in many combinatorial optimization problems and they are building blocks of modern constraint programming languages. It is thus not surprising that `Localizer++` supports them directly as well. Figure 3 depicts a constraint-based statement for the queens problem. The instructions

```
OplLSAllDifferent r(queen);
OplLSAllDifferent ld(queenPos);
OplLSAllDifferent ud(queenNeg);
```

declares the alldifferent constraints. Observe that, contrary to typical constraint programs, it is important to name them since the search procedure makes use of various information on these constraints. The instructions

```
mgr.post(r); mgr.post(ld); mgr.post(ud);
```

post the constraints to the local search manager. The invariant

```
success << (r.isTrue() && ld.isTrue() && ud.isTrue())
```

defines `success` to be true when all three constraints are satisfied in the current state. The specification of the local move remains the same as before, since an expression of the form `alldiff[v]` on an alldifferent constraint `alldiff` represents the violation degree of value `v`.

Observe the resulting simplicity of the declarative component which closely resembles a constraint programming statement of the problem. Of course, the search components are fundamentally different because of the nature of these two approaches. Note also that this implementation is similar in efficiency to the linear implementation presented earlier, since the alldifferent constraints are essentially rewritten into these invariants.

3.5 Experimental Results

We briefly discuss experimental results of the constraint-based statement (similar observations can be made for the second statement as well). As mentioned, the algorithm was expected to run in quadratic time. The experimental results, plotted in Figure 4, confirm the theoretical analysis. The figure shows the running times of the algorithm (red curve) and the function $x^2/350000$ that

```
void queens(int n)
{
   OplLSManager mgr;
   OplIntRange Size(mgr,1,n), SizeP(mgr,2,2*n), SizeN(mgr,1-n,n+1);
   OplIntParameter q(mgr),v(mgr);

   OplIntIncArray queen(mgr,Size,Size,"queen");
   OplUniformDistribution d(mgr,Size);
   for (int k = 1; k <= n ; k++)
      queen[k] = d.get();
   OplIntIncArray queenPos(mgr,Size,SizeP,"queenPos");
   OplIntIncArray queenNeg(mgr,Size,SizeN,"queenNeg");
   OplIntInc success(mgr);

   mgr.post(OplForall(q,Size,queenPos[q] << queen[q] + q));
   mgr.post(OplForall(q,Size,queenNeg[q] << queen[q] - q));

   OplLSAllDifferent r(queen);
   OplLSAllDifferent ld(queenPos);
   OplLSAllDifferent ud(queenNeg);
   mgr.post(r); mgr.post(ld); mgr.post(ud);
   mgr.post(success << (r.isTrue() && ld.isTrue() && ud.isTrue()));

   OplLSAtom move =
      OplAnd(OplSelectMax(q,Size,r[queen[q]]+ld[queenPos[q]]+ud[queenNeg[q]]),
             OplSelectMin(v,Size,r[v]+ld[v+q]+ud[v-q]),
             queen[q] <<= v);
   OplLSSearch search = mgr.newSearch();
   for(int it = 0; it <= 20*n; it++) {
      if (success.getValue()) break;
      search.commit(move);
   }
}
```

Figure 3: The Queens Problem in `Localizer++` (A Constraint-Based Version).

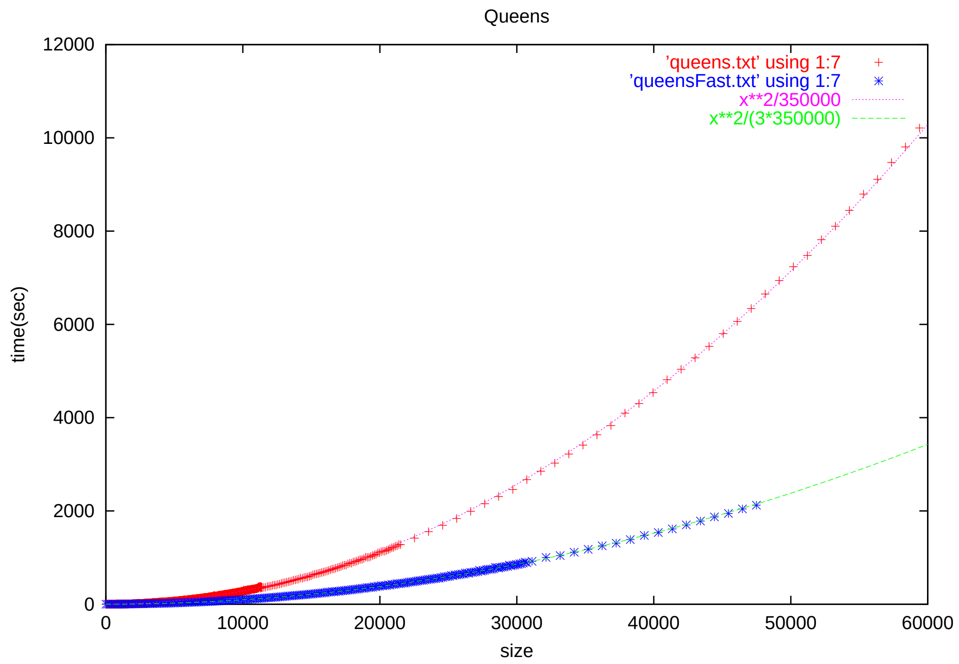

Figure 4: Experimental Results for the Queens Problem.

fits them. Table 3.5 reports the computation times for the constraint-based statement (line `CBS`) in seconds a 800mhz PC running linux. It can be seen that this `Localizer++` statement solves very large instances of this problem quickly. For instance, it takes 2.63 seconds for 1024 queens, about 4.6 hours for 65536 queens and about 56 hours for 262,144 queens.

In fact, a small optimization may further reduce these times. The key idea is to maintain the queens with maximal conflicts incrementally. It suffices to declare the incremental set variable

```
OplIntSetInc MaxConflict(mgr);
```

to declare the new invariant

```
MaxConflict << OplArgMax(q,Size,r[queen[q]]+ld[queenPos[q]]+ud[queenNeg[q]]));
```

and to modify the search procedure to become

```
OplLSAtom move =
   OplAnd(OplSelect(q,MaxConflict),
          OplSelectMin(v,Size,r[v]+ld[v+q]+ud[v-q]),
          queen[q] <<= v);
```

The invariant maintains the set of queens whose number of conflicts is maximal. Observe that the expression to maximize involves arrays of incremental variables which are indexed by incremental variables.[3] The local move simply selects a queen in set `MaxConflict` and chooses a value that minimizes its conflicts. Observe how easy it is to introduce this extension. Figure 4 also plots these results and the function $x^2/(3 * 350000)$ that fits them. The line `ICBS` in Table 3.5 reports the computation times of the improved version in seconds. The statement takes 1.39 seconds for 1024 queens and about 2.8 hours for 65536 queens.

4 The Magic Square Problem

The magic square problem consists of assigning numbers between 1 and n^2 to a square matrix so that all its rows, columns, and upper and lower diagonals sum to the same value and all numbers are distinct. This apparently simple problem is extremely difficult for state-of-the-art mathematical and constraint programming systems. This section presents a tabu search algorithm in `Localizer++` which solves large instances of this problem in reasonable space and time.

The magic square problem illustrates several interesting aspects of `Localizer++`. It introduces equation systems, multidimensional selectors, simulation operators, and counters. Once again, we present several implementations which illustrate different aspects of `Localizer++`. Section 4.1 describes a high-level modeling of the problem, Section 4.2 presents the basic tabu search implementation, Section 4.3 shows how to use a more compact representation of the tabu list, and Section 4.4 shows how to integrate several moves and strategies within a `Localizer++` implementation.

4.1 High-Level Modeling

The tabu search algorithm is expressed in terms of an $n \times n$ matrix. Since the rows, the columns, and the upward and downward diagonals must sum up to

$$T = n \times (n^2 + 1)/2,$$

the problem constraints are given by

$$\begin{cases} \sum_{j=1}^{n} magic[i,j] = T & (1 \leq i \leq n) \\ \sum_{i=1}^{n} magic[i,j] = T & (1 \leq j \leq n) \\ \sum_{i=1}^{n} magic[i,i] = T & \\ \sum_{i=1}^{n} magic[i,n-i+1] = T & \end{cases}$$

The basic idea underlying the local search consists of swapping two entries in the matrix, i.e., to select two pairs (i,j) and (i_1,j_1) and to swap the values of $magic[i,j]$ and $magic[i_1,j_1]$. In addition, progress is measured by the violation degree of the equations. The violation degree of a constraint $l = r$ is given by $\mid l - r \mid$ and the violation degree of an equation system is the summation of the violation degrees of all its equations.

A simple local improvement meta-heuristic is not sufficient for this problem and our algorithm uses a tabu search heuristic to escape local minima. Square elements which are swapped are marked tabu for a number of iterations. In addition, the length of the tabu list is dynamic: it is decreased when a local move improves the violation degree and it is increased when it induces a degradation.

[3]This is equivalent to an `element` constraint in constraint programming [37].

4.2 A Basic Tabu Search Statement

Figure 5 depicts a basic `Localizer++` statement for the magic square. The general structure is similar to the queens problem. It mainly consists of the declarative and search components.

The Declarative Component The first instructions declare the manager, the value `T` mentioned previously, and the parameters and range used in the implementation. The instructions

```
OplIntIncMatrix<2> magic(mgr,Size,Size,"magic");
OplIntIncMatrix<2> tabu(mgr,Size,Size,"tabu");
```

declare two matrices of incremental variables. Matrix `magic` represents the solution. Matrix `tabu` is used to implement the tabu list and memorizes the last iteration where a square element was swapped. The next instructions initialize these two matrices. Observe that the initial configuration simply assigns `magic[i][j]` to `i*(n-1)-j`. The instructions

```
OplSystemEquation E(mgr);
for (int k = 1; k <= n ; k++) {
   E.add(T == OplSum(i,Size,magic[k][i]));
   E.add(T == OplSum(i,Size,magic[i][k]));
}
E.add(T == OplSum(i,Size,magic[i][i]));
E.add(T == OplSum(i,Size,magic[i][n-i+1]));
mgr.post(E);
OplIntInc vd = E.getViolationDegree();
```

are the declarative component of the application. They declare the equation system `E` and the incremental variable `vd` which represents the violation degree of `E` at the current node. Observe the simplicity of this part of the implementation.

The Search Component The rest of the statement describes the search procedure. The instruction

```
OplIntInc tabuLength(mgr,2);
```

declares an incremental variable to represent the length of the tabu list. Its initial value is 2. The instruction

```
OplCounter it(mgr,0);
```

declares a counter. *Counters* are a novel abstraction of `Localizer++` which encapsulates an incremental variable and a variety of data-driven operations. Counters are particularly convenient to implement advanced meta-heuristics, although the basic statement only uses it in a limited way. The next instruction declares the local move and is one of the most interesting features of this model. Let us analyze its parts in some detail. The instruction

```
void magic(int n)
{
   OplLSManager mgr;
   int T = (n * (n*n + 1))/2;
   OplIntParameter i(mgr), j(mgr), e(mgr), i1(mgr), j1(mgr), old(mgr);
   OplIntRange Size(mgr,1,n);
   OplIntIncMatrix<2> magic(mgr,Size,Size,"magic");
   OplIntIncMatrix<2> tabu(mgr,Size,Size,"tabu");
   for (int k = 1; k <= n ; k++)
      for (int l = 1; l <= n ; l++) {
         magic[k][l] = (k-1)*n + l;
         tabu[l][k] = -1;
      }
   OplSystemEquation E(mgr);
   for (int k = 1; k <= n ; k++) {
      E.add(T == OplSum(i,Size,magic[k][i]));
      E.add(T == OplSum(i,Size,magic[i][k]));
   }
   E.add(T == OplSum(i,Size,magic[i][i]));
   E.add(T == OplSum(i,Size,magic[i][n-i+1]));
   mgr.post(E);
   OplIntInc vd = E.getViolationDegree();

   OplIntInc tabuLength(mgr,2);
   OplCounter it(mgr,0);
   OplLSAtom tabumove =
      OplAnd(OplLet(old,vd),
             OplSelectMin(i,Size,
                          j,OplSetOf(e,Size,tabu[i][e] <= it),
                          i1,OplSetOf(e,Size,e>= i),
                          j1,OplSetOf(e,Size,tabu[i1][e]<=it && (i1>i||e>j)),
                          OplLookahead(vd,magic[i][j] ^= magic[i1][j1])),
             magic[i][j] ^= magic[i1][j1],
             tabu[i1][j1] <<= it + tabuLength, tabu[i][j] <<= it + tabuLength,
             OplIf(vd < old && tabuLength > 2,tabuLength <<= tabuLength - 1),
             OplIf(vd >= old && tabuLength < 10,tabuLength <<= tabuLength + 1));
   OplLSSearch search = mgr.newSearch();
   for( ; it.getValue() < 10000; it++) {
      if (vd.getValue() == 0) break;
      search.commit(tabumove);
   }
}
```

Figure 5: A Tabu Algorithm for the Magic Square Problem in `Localizer++` (First Version).

```
OplSelectMin(i,Size,
             j,OplSetOf(e,Size,tabu[i][e] <= it),
             i1,OplSetOf(e,Size,e>= i),
             j1,OplSetOf(e,Size,tabu[i1][e]<=it && (i1>i||e>j)),
             OplLookahead(vd,magic[i][j] ^= magic[i1][j1]))
```

selects the pairs to swap. It illustrates a multidimensional selector as well as a simulation operator. It selects two pairs `(i,j)` and `(i1,j1)` which are not tabu (i.e., their tabu mark is smaller than the current iteration) and which produces the smallest violation degree when `magic[i][j]` and `magic[i1][j1]` are swapped. The instruction

```
magic[i][j] ^= magic[i1][j1])
```

swaps the values of `magic[i][j]` and `magic[i1][j1]`. Instruction `OplLoookahead(E,M)` is a simulation operator. It performs move `M`, evaluates expression `E` in the new state, and undoes the move. It returns the result of the evaluation. The expression `OplSetOf(e,S,C)` return the set of elements `e` in set `S` satisfying condition `C`. The remaining instructions

```
magic[i][j] ^= magic[i1][j1],
tabu[i1][j1] <<= it + tabuLength, tabu[i][j] <<= it + tabuLength,
OplIf(vd < old && tabuLength > 2,tabuLength <<= tabuLength - 1),
OplIf(vd >= old && tabuLength < 10,tabuLength <<= tabuLength + 1));
```

perform the move and update the tabu list. Both pairs `(i,j)` and `(i1,j1)` are marked tabu by specifying that they cannot be selected until iteration `it + tabuLength`. The length of the tabu list is then modified according to whether the move has produced an improvement or a degradation in the violation degree. Note that the instruction `OplLet(old,vd)` had saved the old value of `vd` in `old`. The rest of the statement should not raise much difficulty. A new local search is created and the local move is applied for a number of iterations or until a solution is found. The counter drives the iteration since it is used in the local move. Once again, observe the conciseness of both the declarative and the search parts of this implementation.

4.3 A Sparse Implementation of the Tabu List

The previous implementation uses a matrix to represent the tabu statuses of the elements of the square. Although this is perfectly acceptable for this application (since a matrix is already needed for the solution), many applications require a sparser representation of the tabu list. Figure 6 shows an implementation where the tabu list is implemented with a set. Most of the implementation is left unchanged and we focus only on the differences. Instruction

```
OplTupleSet tabuSet(mgr);
```

declares an empty set of tuples. The multidimensional selector contains expressions of the form

```
OplMember(tabuSet,OplTuple(i,j))
```

```
void magic(int n)
{
   OplLSManager mgr;
   int T = (n * (n*n + 1))/2;
   OplIntParameter i(mgr), j(mgr), e(mgr), i1(mgr), j1(mgr), old(mgr);
   OplIntRange Size(mgr,1,n);
   OplIntIncMatrix<2> magic(mgr,Size,Size,"magic");
   OplTupleSet tabuSet(mgr);
   for (int k = 1; k <= n ; k++)
      for (int l = 1; l <= n ; l++)
         magic[k][l] = (k-1)*n + l;
   OplSystemEquation E(mgr);
   for (int k = 1; k <= n ; k++) {
      E.add(T == OplSum(i,Size,magic[k][i]));
      E.add(T == OplSum(i,Size,magic[i][k]));
   }
   E.add(T == OplSum(i,Size,magic[i][i]));
   E.add(T == OplSum(i,Size,magic[i][n-i+1]));
   mgr.post(E);
   OplIntInc vd = E.getViolationDegree();

   OplIntInc tabuLength(mgr,2);
   OplCounter it(mgr,0);
   OplLSAtom tabumove = OplAnd(
      OplLet(old,vd),
      OplSelectMin(i,Size,
                   j,OplSetOf(e,Size,!OplMember(tabuSet,OplTuple(i,e))),
                   i1,OplSetOf(e,Size,e>= i),
                   j1,OplSetOf(e,Size,!OplMember(tabuSet,OplTuple(i1,e)) && (i1>i||e>j)),
                   OplLookahead(vd,magic[i][j] ^= magic[i1][j1])),
      magic[i][j] ^= magic[i1][j1],
      OplInsert(tabuSet,OplTuple(i,j)), OplInsert(tabuSet,OplTuple(i1,j1)),
      it.whenValue(it+tabuLength,OplRemove(tabuSet,OplTuple(i,j))),
      it.whenValue(it+tabuLength,OplRemove(tabuSet,OplTuple(i1,j1))),
      OplIf(vd < old && tabuLength > 2,tabuLength <<= tabuLength - 1),
      OplIf(vd >= old && tabuLength < 10,tabuLength <<= tabuLength + 1));
   OplLSSearch search = mgr.newSearch();
   for( ; it.getValue() < 10000; it++) {
      if (vd.getValue() == 0) break;
      search.commit(tabumove);
   }
}
```

Figure 6: A Tabu Algorithm for the Magic Square Problem in `Localizer++` (Second Version).

which tests if a pair `(i,j)` is in the tabu set. Marking a pair as tabu simply consists in adding it to the tabu set using instructions of the form

```
OplInsert(tabuSet,OplTuple(i,j))
```

The instructions

```
it.whenValue(it+tabuLength,OplRemove(tabuSet,OplTuple(i,j))),
it.whenValue(it+tabuLength,OplRemove(tabuSet,OplTuple(i1,j1))),
```

are most interesting and are responsible for removing the pairs `(i,j)` and `(i1,j1)` from the tabu list at the appropriate iteration. They use the counter `it` and specify that the pairs `(i,j)` and `(i1,j1)` must be removed from the tabu list when the counter value reaches the summation of its current value and the length of the tabu list. The `whenValue` construct simplifies greatly the management of the tabu list and produces more readable search procedures, since the management of the tabu list is isolated in a couple of instructions.

4.4 Adding a First Improvement Strategy

Many local search algorithms explore a combination of local moves or apply a sequence of meta-heuristics (e.g., [34]) to improve efficiency and/or to produce higher-quality solutions. It is simple to implement these strategies in `Localizer++`. It suffices to define several local moves and to apply them appropriately. To illustrate this feature, we show how to enhance the previous implementation of the magic square by adding a first improvement iteration before starting the tabu search. Figure 7 shows the search component of the new implementation, the declarative component remaining the same. The first part of the figure is most interesting, since it contains the novel instructions. It features a *circular selector*

```
OplSelectCircular(i,Size,
                  j,Size,
                  i1,OplSetOf(e,Size,e>= i),
                  j1,OplSetOf(e,Size,(i1>i||e>j)),
                  vd > OplLookahead(vd,magic[i][j] ^= magic[i1][j1]))
```

which selects the first swap improving the violation degree. The selector fails if there is no such swap, i.e., if there exists no pairs `(i,j)` and `(i1,j1)` such that

```
vd > OplLookahead(vd,magic[i][j] ^= magic[i1][j1]))
```

in which case the local move itself fails. The selector is circular which means that an application of the selector restarts from where the previous successful application exited. More precisely, if the candidate moves are numbered $1, \ldots, m$ and the previous application returned move k, the next application explores the moves in the order $k+1, \ldots, m, 1, \ldots, k$. The instructions

```
int improvement;
do {
```

```
void magic(int n)
{
   ...
   OplLSSearch search = mgr.newSearch();
   OplLSAtom imove =
      OplAnd(OplSelectCircular(i,Size,
                               j,Size,
                               i1,OplSetOf(e,Size,e>= i),
                               j1,OplSetOf(e,Size,(i1>i||e>j)),
                               vd > OplLookahead(vd,magic[i][j] ^= magic[i1][j1])),
             magic[i][j] ^= magic[i1][j1]));
   int improvement;
   do {
      if (vd.getValue() == 0) break;
      improvement = search.commit(imove);
   } while (improvement);

   OplIntInc tabuLength(mgr,2);
   OplCounter it(mgr,0);
   OplLSAtom tabumove =
      OplAnd(OplLet(old,vd),
             OplSelectMin(i,Size,
                          j,OplSetOf(e,Size,!OplMember(tabuSet,OplTuple(i,e))),
                          i1,OplSetOf(e,Size,e>= i),
                          j1,OplSetOf(e,Size,!OplMember(tabuSet,OplTuple(i1,e)) && (i1>i||e>j)),
                          OplLookahead(vd,magic[i][j] ^= magic[i1][j1])),
             magic[i][j] ^= magic[i1][j1],
             OplInsert(tabuSet,OplTuple(i,j)), OplInsert(tabuSet,OplTuple(i1,j1)),
             it.whenValue(it+tabuLength,OplRemove(tabuSet,OplTuple(i,j))),
             it.whenValue(it+tabuLength,OplRemove(tabuSet,OplTuple(i1,j1))),
             OplIf(vd < old && tabuLength > 2,tabuLength <<= tabuLength - 1),
             OplIf(vd >= old && tabuLength < 10,tabuLength <<= tabuLength + 1));

   for( ; it.getValue() < 10000; it++) {
      if (vd.getValue() == 0) break;
      search.commit(tabumove);
   }
}
```

Figure 7: The Magic Square Problem in Localizer++ (Third Version).

Tens	1	2	3	4	5	6	7	8	9	10
10	8.16	82.92	13.60	7.52	68.33	50.82	44.80	26.31	224.93	313.23
20	50.71	354.47	280.29	1253.27	258.64	261.60	510.37	379.55	611.14	1969.51
30	622.92	837.78	1128.40	3646.88	776.03	921.28	7031.08	1825.64	3127.57	8553.94
40	2876.31	4113.90	13875.46	4596.70	11805.07	8340.09	10499.32	3497.48	8288.19	23158.58

Table 2: Computation Times for Magic Square in Seconds.

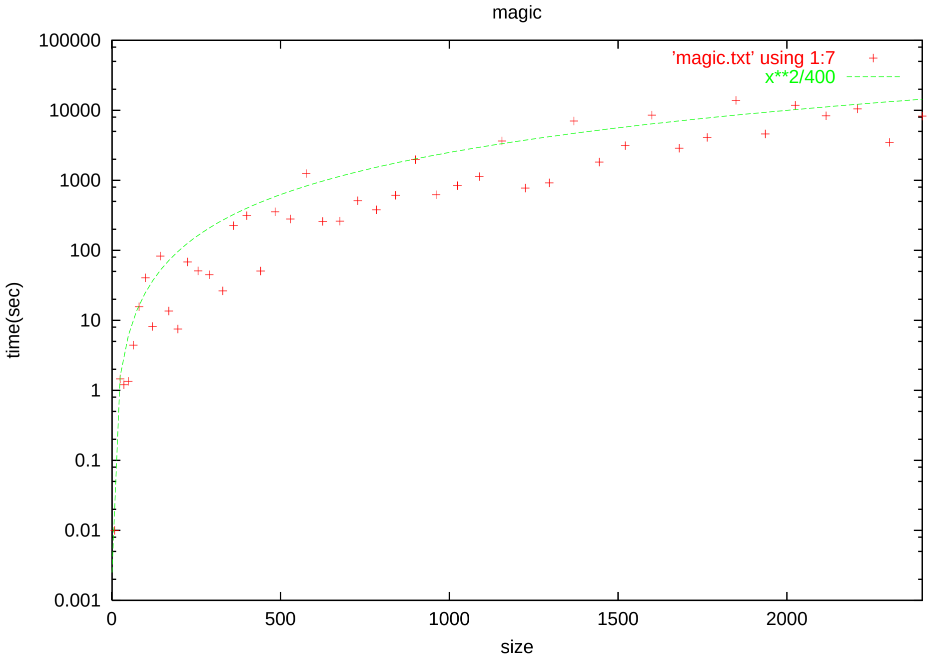

Figure 8: Graphical Display of the Experimental Results for Magic Square.

```
   if (vd.getValue() == 0) break;
   improvement = search.commit(imove);
} while (improvement);
```

simply apply the improvement move until it fails. The algorithm then turns to the tabu search presented earlier.

4.5 Experimental Results

We briefly report experimental results on the last `Localizer++` statement for magic squares of size 11×11 to sizes 50×50 (2500 integer variables with 2500 possible values). Table 4.5 gives the computation times in seconds on a 800mhz PC running linux and Figure 8 plots these results with a log scale. As can be seen, the running time is essentially quadratic in the number of variables and fit the function $X^2/400$. There are however much more variations in the times than for the queens problems, some instances being much harder than others. The statement solves the 11×11 magic square in 8.16 seconds, the 31×31 magic square in about 10 minutes, and the 50×50 square in about 6.5 hours. It is of course possible to solve larger instances of this problem since the algorithm runs in space linearly proportional to the number of variables.

5 The Graph-Partitioning Problem

This section describes our last application, the graph-partitioning problem, and presents an algorithm based on a linear neighborhood inspired by [8]. This graph-partitioning algorithm is very interesting, since it illustrates *incremental set variables* and sophisticated invariants involving these variables as well as cross-products and dynamic filters. It also shows how to implement the large-scale neighborhood heuristic of Kernighan & Lin [17] in a simple way.

5.1 High-Level Modeling

Given a graph $G = (V, E)$, the graph-partitioning problem consists of finding a partition of the vertices V in two sets V_0 and V_1 of equal size in order to minimize the number of crossings, i.e., the number of edges (v_0, v_1) in E such that $v_0 \in V_0$ and $v_1 \in V_1$.

The key idea of the approach is to keep the size of the neighborhood manageable in order to tackle very large partitioning problems. As a consequence, the algorithm does not consider all pairs of vertices $v_0 \in V_0$ and $v_1 \in V_1$ for swapping. It only considers those vertices that appear most promising in V_0 and V_1. More precisely, the algorithm computes the set of vertices in V_0 (resp. V_1) producing the smallest number of crossings when swapped to V_1 (resp. V_0). Observe that there may be several vertices producing this smallest number. In order to maintain these sets efficiently, the algorithm maintains the internal and external costs of each vertex as well as their difference as is traditional in partitioning algorithms (e.g., [28]). Assuming that the partitions are numbered 0 and 1 and that $x[v]$ represents the partition of vertex `v`, the algorithm maintains

$$\begin{array}{lcl} ext[v] & = & \sum_{(v,w)\in E}(x[v] \neq x[w]) \\ inte[v] & = & \sum_{(v,w)\in E}(x[v] = x[w]) \\ diff[v] & = & ext[v] - inte[v] \end{array}$$

Finding the promising vertices to swap can be obtained using the formula ($0 \leq i \leq 1$)

$$\begin{array}{lcl} P[i] & = & \{v \in V \mid x[v] = i\} \\ maxP[i] & = & \max_{v\in P[i]} diff[v] \\ argmaxP[i] & = & \{v \in P[i] \mid diff[v] = maxP[i]\} \end{array}$$

The swaps considered by the algorithm are the pairs from the cross product of $P[0]$ and $P[1]$. The algorithm selects the best swap, i.e., the swap producing the smallest number of crossings. Note that the number of crossings is given by the formula

$$\sum_{(v,w)\in E} (x[v] \neq x[w]).$$

The meta-heuristic used in the algorithm is the Kernighan & Lin's large-scale neighborhood technique. The basic idea is to consider a sequence of successive swaps

$$(v_1, w_1), (v_2, w_2), \ldots, (v_n, w_n)$$

where all v_i and w_i are different, to select the subsequence

$$(v_1, w_1), \ldots, (v_k, w_k)$$

producing the smallest number of crossings, and to define the next state as the node obtained by applying this subsequence.

5.2 The `Localizer++` Statement

Figures 9 and 10 depict the implementation of the local search algorithm in `Localizer++`. Figure 9 mainly describes the declarative component, while Figure 10 shows the search procedure.

The Declarative Component The statement receives as inputs the local search manager, the number of vertices, the number of edges, the adjacency matrix, the set of edges, and the adjacency lists for the vertices. It declares essentially the same objects as those used in the high-level modeling description of the local search. In particular, it declares new types of incremental variables. The instruction

```
OplIntSetIncArray P(mgr,Sets,"P");
```

declares an array of two incremental set variables whose values are sets of integers. These incremental variables are used subsequently in complex invariants filtering, or iterating over, their elements. The instruction

```
OplTupleSetInc Swaps(mgr);
```

declares an incremental variable whose value is a set of tuples. It is used to store the set of pairs (v_0, v_1) considered for swapping. Once again, complex invariants will be built on top of this incremental variable. The instructions

```
OplIntIncArray mark(mgr,V,"mark");
OplCounter marker(mgr,0);
```

declare an array `mark` of incremental variables to mark the vertices and a counter `marker`. They are used to implement the Kernighan & Lin meta-heuristic. Observe that the statement declares an array of set of integers `PP`. Set `PP[i]` will be used to store the elements of `P[i]` which are not marked. The statement then states a number of invariants of various complexities. It is useful to examine some of them, since they illustrate some interesting functionalities of `Localizer++`. The invariant

```
ext[i] << OplSum(e,adjl[i],x[e->get(0)] != x[e->get(1)])
```

```
void partition(OplLSManager mgr,int n,int m,
               OplIntMatrix<2> adj,OplTupleSet E,OplTupleSetArray adjl)
{
   OplIntRange V(mgr,1,n), E(mgr,1,m), Sets(mgr,0,1);
   int h = n/2;
   OplIntIncArray x(mgr,V,"x");
   OplUniformDistribution distr(mgr,Sets);
   for(int i = 1; i <= h ; i++) {
      x[i] = distr.get(); x[i+h] = 1 - x[i].getValue();
   }
   OplIntParameter I(mgr);
   OplTupleParameter e(mgr), s(mgr);

   OplIntIncArray ext(mgr,V,"ext"), inte(mgr,V,"inte"), diff(mgr,V,"diff");
   OplIntSetIncArray P(mgr,Sets,"P"), PP(mgr,Sets,"PP"), argmaxP(mgr,Sets,"argmaxP");
   OplTupleSetInc Swaps(mgr), OplTupleSetInc bestSwaps(mgr);
   OplIntInc total(mgr);
   OplIntIncArray mark(mgr,V,"mark");
   OplCounter marker(mgr,0);

   for(int i = 1; i <= n ; i++) {
      mgr.post(ext[i] << OplSum(J,adjl[i],x[e->get(0)] != x[e->get(1)]));
      mgr.post(inte[i] << OplSum(J,adjl[i],x[e->get(0)] ==x[e->get(1)]));
      mgr.post(diff[i] << ext[i] - inte[i]);
   }
   mgr.post(OplIndicesOf(x,P));
   for(int i = 0; i <= 1 ; i++) {
      mgr.post(PP[i] << OplSetOf(I,P[i],mark[I]<=marker));
      mgr.post(argmaxP[i] << OplArgMax(I,PP[i],diff[I]));
   }
   mgr.post(Swaps << OplCross(argmaxP[0],argmaxP[1]));
   mgr.post(bestSwaps <<
      OplArgMax(s,Swaps,diff[s->get(0)]+diff[s->get(1)]-2*adj[s->get(0)][s->get(1)]));
   mgr.post(total << OplSum(e,E,(x[e.get(0)] != x[e.get(1)]));
   ...
}
```

Figure 9: The Graph-Partitioning Implementation in `Localizer++` (Part I).

```
void partition(OplLSManager mgr,int n,int m,
               OplIntMatrix<2> adj,OplTupleArray e,OplTupleSetArray al)
{
   ...;
   for(int i = 1; i <= h ; i++)
      mark[i] = 0;
   OplLSAtom move = OplSelectBestInSequence(n/4,total,
      OplAnd(
         OplSelect(s,bestSwaps),
         x[s->get(0)] <<= !x[s->get(0)],
         x[s->get(1)] <<= !x[s->get(1)],
         mark[s->get(0)] <<= marker + 1,
         mark[s->get(1)] <<= marker + 1));
   int found;
   do {
      found = search.commit(move);
      marker++;
   } while (found);
}
```

Figure 10: The Graph-Partitioning Implementation in `Localizer++` (Part II).

maintains the external cost of a vertex as the number of edges in its adjacency list whose vertices are not in the same partition. Observe that the set `adjl[i]` is constant while the condition involves incremental variables whose values may change. The invariant

```
OplIndicesOf(x,P)
```

maintains in `P[i]` the set of vertices whose partition is `i`. It could be rewritten into

```
OplForall(i,Sets,P[i] << OplSetOf(j,V,x[j] == i))
```

Some of the remaining invariants are particularly interesting. The invariant

```
PP[i] << OplSetOf(I,P[i],mark[I]<=marker)
```

maintains the set of "unmarked" elements in `P[i]`. Since `P[i]` is an incremental set variable and since the condition `mark[I]<=marker` involves incremental variables, this invariant is non-trivial to maintain incrementally. It needs to be updated each time elements are added or removed from `P[i]` and each time `mark[I]` or `marker` are updated. The invariant

```
argmaxP[i] << OplArgMax(I,PP[i],diff[I])
```

also illustrates well the expressiveness of `Localizer++`. It computes the set `argmaxP[i]` described in the high-level modeling, i.e., `argmaxP[i]` computes the set of elements `I` in `PP[i]` whose value `diff[I]` is maximal among the elements of `PP[i]`. Since `PP[i]` is an incremental set variables and since `diff` is an array of incremental variables, this invariant is once again not obvious to maintain incrementally. The invariant

```
Swaps << OplCross(argmaxP[0],argmaxP[1]));
```

maintains the set of swaps as the cross-product of `argmaxP[0]` and `argmaxP[1]`. Set `Swaps` is an incremental set variable whose values are pairs as mentioned earlier. Finally, the invariant

```
bestSwaps <<
   OplArgMax(s,Swaps,diff[s->get(0)]+diff[s->get(1)]-2*adj[s->get(0)][s->get(1)])
```

maintains `bestSwaps`, an incremental set variable whose value is the subset of swaps with maximal gain.

It is useful to stress the expressiveness of `Localizer++` at this point. The invariants in the graph-partitioning statement are not easy to maintain incrementally and their implementation requires significant programming effort. Their availability in `Localizer++` greatly reduces the development time of this statement and let users focus on high-level aspects of the problem.

The Search Component As mentioned, Figure 10 depicts the search component which implements Kernighan & Lin meta-heuristic. The move is specified by the instruction

```
OplSelectBestInSequence(n/4,total,
   OplAnd(
      OplSelect(s,bestSwaps),
      x[s->get(0)] <<= !x[s->get(0)],
      x[s->get(1)] <<= !x[s->get(1)],
      mark[s->get(0)] <<= marker + 1,
      mark[s->get(1)] <<= marker + 1));
```

which features the operator `OplSelectBestInSequence`. Given an integer `n`, an expression `E`, and a move `M`, operator `OplSelectBestInSequence(n,E,M)` applies the move `M` `n` times to obtain the states $S_1, \ldots, S_n$. Expression `E` is evaluated in each of these states to produce the values $e_1, \ldots, e_n$. The state S_i with the best value e_i becomes the next state. In the graph-partitioning application, this operator is applied to a move which selects one of the best swaps, performs the swaps, and marks the swapped vertices so that they are not selected again in the sequence. Indeed, since the vertices are now marked, they do not appear in sets `PP[i]` (`i=0,1`). Observe that `bestSwaps` is updated incrementally and automatically once vertices are marked. Note also that operator `OplSelect` is stochastic in nature, i.e., it selects a random element from its second argument. The rest of the search procedure is the loop

```
int found;
do {
   found = search.commit(move);
   marker++;
} while (found);
```

Observe how the counter is incremented which has the effect of "unmarking" all the vertices for the next iteration. Once again, it is worthwhile stressing the expressiveness of `Localizer++`: it is rather simple to design an efficient implementation of the meta-heuristics.

5.3 Experimental Results

We now report experimental results on the graph-partitioning problem. Once again, it is important to stress that the goal of this paper was not to show the best possible graph-partitioning model that can be written in `Localizer++` but to illustrate the functionalities of `Localizer++`. It is certainly possible to write `Localizer++` statements producing higher-quality solutions and/or running more efficiently. The main value of these results is to validate the approach and to show that a high-level library for local search can be a practical and valuable tool.

Table 3 depicts experimental results on the graph-partitioning problems. It gives results for the statement just presented as well as the `Localizer++` implementation of the simulated annealing of [15]. This simulated annealing statement is comparable in performance to the specialized algorithm (this was already the case for `Localizer` [22]). Note that the two statements use fundamentally different neighborhoods: the algorithm in [15] swaps vertices from one set to the other, does not maintain a balanced partition, and uses a penalty term in the objective function to drive the search toward balanced solutions. The columns in the table report the number of vertices, the density of

```
class OplArrayInvariantI : public OplInvariantI {
   void OplArrayInvariantI(OplIntIncArray trg,OplIntIncArray src);
   virtual void eval() = 0;
   virtual void incrEval(int k) = 0;
};
```

Figure 11: The API of Array Invariants.

the graph and, for each algorithm, the value of the best found solution, the average value of the solutions and their standard deviation, the average computation time of the solutions and their standard deviation. Each line in table represents 100 runs of the two algorithms on a given graph (since the algorithms are randomized).

Interestingly, the Kernighan & Lin meta-heuristic on the linear neighborhood behaves very well. On our experiments, it generally produces higher-quality solutions than simulated annealing. Its computation times are slower than simulated annealing, although they remain clearly acceptable. Once again, the goal here is not report results on graph-partitioning algorithms but to show that a library like `Localizer++` is a valid addition to the set of optimization tools.

6 Extending the Library

As mentioned, `Localizer++` is an extensible library: users may add their own invariants, constraints, and search procedures. This section reviews the basic principles to extend the library.

6.1 Adding Invariants

New invariants can be added to `Localizer++` by subclassing a number of abstract classes. Once again, it is outside the scope of this paper to give a comprehensive presentation of these classes. This section simply illustrates how to add a new array invariant, since other types of invariants may be added in a similar fashion. Figure 11 depicts the interface of array invariants. The abstract class contains a constructor and two virtual pure methods. The constructor expects the target array (i.e., the array maintained by the invariant) and the source array (i.e., the array used by the invariant). Virtual method `eval` is called for the first evaluation of the invariant, while virtual method `incrEval(int k)` is called to update the invariant when element `src[k]` is modified. The `Localizer++` implementation takes care of calling these methods at appropriate times to guarantee the correctness of the propagation.

Figure 12 shows how to implement invariant `OplCardIndicesOf`, used in the queens problem, as an array invariant. Recall that this invariant maintains `_card[i]` as the number of occurrences of `i` in array `_vars`. Method `eval` simply initializes the cardinalities and then increments them appropriately by iterating over array `_vars`. The incremental method is more interesting. When

Benchmark			Kernighan & Lin					Simulated Annealing				
Bench	Sz	D.	Best	$\mu(S)$	$\sigma(S)$	$\mu(T)$	$\sigma(T)$	Best	$\mu(S)$	$\sigma(S)$	$\mu(T)$	$\sigma(T)$
1	124	5	66	72.88	3.28	0.54	0.21	68	76.64	4.44	0.07	0.01
2	124	5	52	57.03	2.84	0.42	0.11	53	60.49	3.96	0.07	0.01
3	124	5	58	62.80	5.07	0.41	0.12	58	68.67	6.05	0.07	0.01
4	124	5	60	65.63	3.52	0.47	0.16	60	69.71	3.88	0.07	0.01
5	124	5	55	60.87	3.95	0.45	0.15	55	66.09	4.85	0.07	0.01
1	124	10	185	189.78	3.76	0.38	0.10	185	197.79	10.42	0.08	0.01
2	124	10	190	195.60	3.02	0.36	0.09	192	201.68	7.52	0.08	0.01
3	124	10	191	195.75	3.90	0.37	0.09	191	204.39	11.18	0.09	0.01
4	124	10	192	197.83	3.98	0.36	0.09	191	204.80	7.65	0.08	0.01
5	124	10	181	187.30	4.16	0.36	0.09	181	193.45	6.51	0.08	0.01
1	124	20	454	465.45	5.59	0.48	0.12	457	492.09	34.67	0.12	0.01
2	124	20	445	451.37	4.65	0.48	0.11	445	467.76	23.36	0.12	0.01
3	124	20	450	457.50	5.23	0.46	0.10	451	476.06	28.61	0.12	0.01
4	124	20	443	452.38	6.72	0.48	0.11	444	475.09	33.76	0.12	0.01
5	124	20	448	454.07	6.69	0.51	0.12	450	477.24	27.80	0.12	0.01
1	250	5	107	120.63	7.68	2.91	1.01	111	129.80	8.01	0.17	0.03
2	250	5	114	125.58	5.09	3.60	1.14	117	133.03	5.90	0.16	0.03
3	250	5	124	132.32	4.37	1.94	0.58	125	138.78	5.92	0.18	0.03
4	250	5	117	130.14	5.18	2.81	0.86	122	135.69	5.77	0.16	0.02
5	250	5	123	133.55	5.09	2.61	0.97	125	137.24	5.91	0.17	0.03
1	250	10	348	362.58	7.59	1.43	0.41	355	374.72	9.03	0.20	0.03
2	250	10	350	365.74	6.60	1.55	0.47	352	375.79	8.50	0.19	0.03
3	250	10	327	339.81	7.62	1.58	0.41	334	352.20	8.47	0.19	0.03
4	250	10	346	356.68	6.01	1.52	0.44	344	365.97	8.77	0.19	0.03
5	250	10	345	358.50	5.96	1.44	0.37	349	368.27	9.25	0.19	0.03
1	250	20	876	894.59	8.94	1.54	0.37	885	913.63	17.14	0.25	0.03
2	250	20	868	883.86	9.09	1.44	0.41	872	901.01	29.01	0.26	0.03
3	250	20	858	870.27	7.12	1.45	0.28	865	890.59	15.17	0.25	0.03
4	250	20	885	899.38	9.07	1.52	0.39	885	915.73	20.24	0.26	0.03
5	250	20	835	847.35	11.02	1.57	0.40	836	869.16	18.44	0.25	0.03
1	500	5	255	273.78	8.64	32.74	11.81	265	284.78	9.50	0.41	0.08
2	500	5	249	261.00	6.83	35.27	14.21	247	268.69	8.84	0.41	0.08
3	500	5	234	249.24	7.38	27.50	8.71	244	258.47	7.21	0.41	0.07
4	500	5	230	244.78	6.27	22.57	8.21	233	254.45	9.91	0.40	0.07
5	500	5	211	232.02	9.05	34.22	11.38	228	241.65	7.33	0.40	0.06
1	500	10	662	680.25	9.94	8.40	2.60	668	700.06	13.59	0.47	0.09
2	500	10	675	696.53	10.68	7.21	2.27	692	714.69	12.63	0.44	0.06
3	500	10	697	717.37	9.35	7.67	2.37	711	739.41	12.67	0.47	0.09
4	500	10	691	713.08	12.10	8.57	2.72	708	733.35	11.23	0.44	0.07
5	500	10	717	737.45	11.90	8.36	2.42	729	755.65	11.08	0.45	0.06
1	500	20	1764	1799.47	16.73	5.22	1.37	1783	1831.75	20.02	0.60	0.09
2	500	20	1702	1733.69	15.55	5.72	1.43	1737	1771.29	17.79	0.57	0.06
3	500	20	1780	1806.84	14.49	6.13	1.84	1806	1838.12	17.63	0.60	0.08
4	500	20	1715	1755.22	16.36	5.91	1.50	1745	1790.45	20.04	0.61	0.06
5	500	20	1741	1766.61	15.96	5.55	1.68	1766	1794.22	15.65	0.60	0.08

Table 3: Experimental Results on Graph Partitioning.

```
class OplCardIndicesOfI : public OplArrayInvariantI {
   OplIntIncArray _cards;
   OplIntIncArray _vars;
public:
   OplCardIndicesOfI(OplIntIncArray cards,OplIntIncArray vars)
      :  OplArrayInvariantI(card,vars),_cards(cards),_vars(vars) {}
   void eval() {
      const int lowVars = _vars.getRange().getLow();
      const int upVars = _vars.getRange().getUp();
      const int lowCards = _cards.getRange().getLow();
      const int upCards = _cards.getRange().getUp();
      for(int i=lowCards;i<=upCards;i++)
         _cards[i] = 0;
      for(int i=lowVars;i<=upVars;i++)
         _cards[_vars[i].getValue()]++;
   }
   void incrEval(int k) {
      _cards[_vars[k].getValue()]++;
      _cards[_vars[k].getOld()]--;
   }
};
```

Figure 12: The Implementation of the Invariant `OplCardIndicesOf`.

```
class OplLSUserAtom {
   OplLSUserAtomI* _impl;
public:
   OplLSUserAtom(OplLSManager mgr);
   virtual void apply(OplLSSearch) = 0;
};
```

Figure 13: The API of Search Procedures.

method `incrEval` is called with an actual parameter `k`, it means that `_vars[k]` was updated since the last propagation of the invariants. To perform the necessary incremental update, `Localizer++` gives access to the last two values of `_vars[k]`: its old value `o`, i.e., its value at the end of the last propagation step, and its current value `v`. With these two values at hand, it is simple to implement the incremental method: it suffices to decrement `_cards[o]` and to increment `_cards[v]`. This is exactly what the implementation in Figure 12 does.

6.2 Adding Search Procedures

New search constructs can also be added to `Localizer++` by extending `OplLSUserAtom`, the abstract class depicted in Figure 13. This class has only one virtual pure method `apply` which specifies the operational behavior of the construct. Method `apply` receives, as argument, a search object which offers a number of functionalities to implement complex constructs. We use three of them here: method `pushNode()` which creates a new node in the search space, method `apply` which executes a move, and method `popNode` which returns to the previous node in the search.[4] Once these methods are available, it becomes relatively easy to implement complex search constructs. To illustrate this functionality, Figure 14 shows how to implement a simulation operator `OplLookaheadI(v,exp,move)` which assigns to parameter `v` the value of `exp` after execution of `move`. The `apply` method first pushes a new node, applies the move, evaluates the expression in the new state and, finally, pops the created node to restore the initial state.

7 Conclusion

This paper presented the design of `Localizer++`, an extensible object-oriented library for local search. `Localizer++` is based on a two-level architecture that supports both declarative abstractions to describe the neighborhood and high-level search constructs to specify local moves and meta-heuristics. The declarative abstractions, which include invariants and constraints, automate the maintenance of incremental data structures, one of the most tedious and error-prone aspects of

[4]These methods are somewhat related to the creation of choice points and to backtracking in mathematical and constraint programming systems.

```
class OplLookaheadI : public OplLSUserAtom {
   OplIntParameter _v;
   OplIntInc _exp;
   OplLSAtom _move;
public:
   OplLookaheadI(OplIntParameter v,OplIntInc exp,OplLSAtom move)
      :  OplLSUserAtom(move.getMgr()), _v(v), _exp(exp), _move(move) {}
   void apply(OplLSSearch search) {
      search.pushNode();
      search.apply(_move);
      _v = _expr->getValue();
      search.popNode();
   }
};
```

Figure 14: The Implementation of `OplLookahead`.

local search algorithms. The search constructs, which include multidimensional selectors, simulation operators, and counters, may greatly simplify the specification of local moves and meta-heuristics. In addition, `Localizer++` inherits from `OPL++` many of the functionalities typically found only in modeling languages, such as multidimensional arrays, sparse data structures, and high-level set, algebraic, and logical operators. Finally, `Localizer++` is an open system which makes it easy to add new invariants, new constraints, and new search constructs. The paper illustrated these features on a number of applications, focusing on showing the expressiveness and the versatility of `Localizer++`. Preliminary experimental results also indicate that `Localizer++` retains much of the space and time efficiency of special-purpose implementations.

There are, of course, many opportunities for future work. They include research on high-level invariants and constraints, more advanced search constructs, and integration with local search frameworks. In addition, since `OPL++` now supports both a constraint programming and a local search library, it seems natural to study how to best support integrations of local search and constraint programming [29, 35].

Acknowledgments

Special thanks to Philippe Codognet for pointing out the magic square problem to us.

References

[1] E. Aarts and J.K. Lenstra. *Local Search in Combinatorial Optimization.* Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley & Sons Ltd, England, 1997.

[2] A. Andreatta. A framework for the development of local search heuristics with an application to the phylogeny problem, 1997.

[3] J. Bisschop and A. Meeraus. On the Development of a General Algebraic Modeling System in a Strategic Planning Environment. *Mathematical Programming Study*, 20:1–29, 1982.

[4] A. Colmerauer. Spécification de Prolog IV. Technical report, Laboratoire d'informatique de Marseille, 1996.

[5] B. De Backer, V. Furnon, and P. Shaw. An object model for meta-heuristic search in constraint programming. In *Proceedings of the First International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'99)*, Ferrara, Italy, 1999.

[6] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December 1988.

[7] H. El Sakkout and M. Wallace. Probe Backtrack Search for Minimal Perturbation in Dynamic Scheduling. *Constraints*, 5(4), October 2000.

[8] C.M. Fiduccia and R.M. Mattheyses. A Linear-Time Heuritics for Improving Network Partitions. In *Proceedings of the ACM IEEE nineteenth Design Automation Conference*, pages 175–181, Los Alamitos, Ca, August 1982. IEEE Computer Society Press.

[9] A. Fink, S. Voss, and D. Woodruff. Building reusable software components for heuristc search, 1998.

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

[11] P. Galinier and J.-K. Hao. A General Approach for Constraint Solving by Local Search. In *Proceedings of the Second International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'00)*, Paderborn, Germany, March 2000.

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

[13] F. Glover and M. Laguna. *Tabu Search.* Kluwer Academic Publishers, 1997.

[14] Ilog Solver 4.4. Reference Manual. Ilog SA, Gentilly, France, 1998.

[15] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon. Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning. *Operations Research*, 37(6):865–893, 1989.

[16] 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.

[17] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49:291–307, 1970.

[18] Kindervater, G. and Savelsbergh, M.W. Vehicle routing: Handling edge exchanges. In E. Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 482–520. Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley & Sons Ltd, England, 1997.

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

[20] F. Laburthe and Y. Caseau. SALSA: A Language for Search Algorithms. In *Fourth International Conference on the Principles and Practice of Constraint Programming (CP'98)*, Pisa, Italy, October 1998.

[21] Dell'Amico M. and Trubian M. Applying Tabu Search to the Job-Shop Scheduling Problem. *Annals of Operations Research*, 41:231–252, 1993.

[22] L. Michel. *Localizer: A Modeling Language for Local Search.* PhD thesis, Brown University, October 1998.

[23] L. Michel and P. Van Hentenryck. Localizer: A Modeling Language for Local Search. In *Third International Conference on the Principles and Practice of Constraint Programming (CP'97)*, Lintz, Austria, October 1997.

[24] L. Michel and P. Van Hentenryck. Localizer: A Modeling Language for Local Search. *Informs Journal on Computing*, 11(1):1–14, 1999.

[25] L. Michel and P. Van Hentenryck. Localizer. *Constraints*, 5:41–82, 2000.

[26] L. Michel and P. Van Hentenryck. OPL++: A Modeling Layer for Constraint Programming Libraries. Technical Report CS-00-07, Brown University, December 2000.

[27] 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.

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

[29] G. Pesant, M. Gendreau, and J-M. Rousseau. GENIUS-CP: A Generic Single-Vehicle Routing Algorithm. In *Third International Conference on the Principles and Practice of Constraint Programming (CP'97)*, pages 420–434, Lintz, Austria, 1997.

[30] G. Ramalingam. *Bounded Incremental Computation.* PhD thesis, University of Wisconsin-Madison, 1993.

[31] G. Ramalingam and T. Reps. On competitive on-line algorithms for the dynamic priority-ordering problem. *Information Processing Letters*, 51:155–161, 1994.

[32] A. Schaerf, M. Lenzerini, and M. Cadoli. Local++: A c++ framework for local search algorithms. Technical Report 11-99, Dipartimento di Informatica e Sistemistica, Universita di Roma, La Sapienza, May 1999.

[33] C. Schulte. Programming Constraint Inference Engines. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, volume 1330, pages 519–533, Schloss Hagenberg, Linz, Austria, October 1997. Springer-Verlag.

[34] B. Selman and H. Kautz. Domain-Independent Extensions to GSAT: Solving Large Structured Satisfiability Problems. In *Proceedings of IJCAI-93*, 1993.

[35] P. Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In *Fourth International Conference on the Principles and Practice of Constraint Programming (CP'98)*, pages 417–431, Pisa, Italy, October 1998.

[36] G. Smolka. The Oz Programming Model. In Jan van Leeuwen, editor, *Computer Science Today.* LNCS, No. 1000, Springer Verlag, 1995.

[37] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

[38] P. Van Hentenryck. *The OPL Optimization Programming Language.* The MIT Press, Cambridge, Mass., 1999.