

A Constraint-Based Architecture for Local Search

Laurent Michel
Brown University, Box 1910,
Providence RI 02912
ldm@cs.brown.edu

Pascal Van Hentenryck
Brown University, Box 1910,
Providence RI 02912
pvh@cs.brown.edu

ABSTRACT

Combinatorial optimization problems are ubiquitous in many practical applications. Yet most of them are challenging, both from computational complexity and programming standpoints. Local search is one of the main approaches to address these problems. However, it often requires sophisticated incremental algorithms and data structures, and considerable experimentation. This paper proposes a constraint-based, object-oriented, architecture to reduce the development time of local search algorithms significantly. The architecture consists of declarative and search components. The declarative component includes *invariants*, which maintain complex expressions incrementally, and *differentiable objects*, which maintain properties that can be queried to evaluate the effect of local moves. Differentiable objects are high-level modeling concepts, such as constraints and functions, that capture combinatorial substructures arising in many applications. The search component supports various abstractions to specify heuristics and meta-heuristics. We illustrate the architecture with the language COMET and several applications, such as car sequencing and the progressive party problem. The applications indicate that the architecture allows for very high-level modeling of local search algorithms, while preserving excellent performance.

Categories and Subject Descriptors

D.1.5 [Programming Techniques]: Object-oriented Programming; D.2.13 [Software Engineering]: Reusable Software—*Reusable libraries, Reuse models*; D.3.3 [Programming Languages]: Language Constructs and Features—*Constraints, Frameworks*

General Terms

Algorithms, Experimentation, Languages

Keywords

Constraint, local search, incremental algorithms, combinatorial optimization

1. INTRODUCTION

Combinatorial optimization problems are ubiquitous in practical applications such as logistics, scheduling, resource allocation, and computational biology to name only a few. Most of these problems are NP-complete and challenging both from computational and software engineering standpoints. Indeed, reasonable solutions to these problems often involve complex algorithms and data structures, as well as significant experimentation.

The last two decades have witnessed the emergence of many languages and libraries for combinatorial optimization (e.g., [6, 9, 14, 25, 26]). These languages may dramatically reduce development time for these applications, while inducing small overheads in efficiency. However, most tools focus on global search which includes branch & bound and constraint satisfaction algorithms. They offer little or no support for local search.

Local search is one of the most widely used approaches to combinatorial optimization because it often produces high-quality solutions in reasonable time. Local search tackles combinatorial optimization problems by moving from a configuration to one of its neighbors until a feasible configuration or an optimal configuration is found. The implementation of local search algorithms raises fundamentally new challenges. Indeed, these algorithms generally maintain sophisticated data structures in order to decide where to move next efficiently. These data structures do not evolve too much when moving from configurations to configurations, since neighbors have closely related states. It is thus critical to maintain them incrementally. In addition, local search algorithms also require good exploration strategies (called meta-heuristics) to explore the neighborhood graph. Since the choice of a good neighborhood and meta-heuristic is still an art which may involve considerable experimentation, it is clear that novel abstractions for local search may bring significant benefits. Recent research has started addressing these challenges. But, in general, these proposals (e.g., [2, 8, 22]) restrict attention to frameworks that make it easy to experiment with different meta-heuristics. They do not address high-level modeling issues and incrementality.

This research presents a constraint-based, object-oriented, architecture for local search. The architecture consists of two main components: a declarative component which models the application in terms of constraints and functions, and a search component which specifies the meta-heuristic.

Constraints are a natural vehicle to express combinatorial optimization problems as past research has shown. However, they play a fundamentally novel role in this architecture. Instead of pruning the search space as is the case in global search algorithms, constraints here are *differentiable objects*: They maintain a number of properties incrementally and they provide algorithms to evaluate the effect of various operations on these properties. (Typical properties include the violation degree of a constraint and the set of variables violating the constraint.) As a consequence, although they are used to model the combinatorial problem in a declarative fashion, constraints also encapsulate sophisticated incremental algorithms and data structures. The search component then uses these functionalities to guide the local search.

The resulting architecture has a number of fundamental benefits. Perhaps the most significant feature is to allow practitioners to focus on high-level modeling issues and to relieve them from many tedious and error-prone aspects of local search. The resulting programs are often intuitive and natural, yet they are also efficient since they encapsulate years of research on incremental algorithms. Another important benefit of the architecture is its compositionality. It is easy to add new constraints, and to modify or remove existing ones, without having to worry about the global effect of these changes. In the same spirit, the architecture clearly separates the modeling and search components and makes it easy to experiment with different meta-heuristics without affecting the problem modeling. Finally, the architecture is non-intrusive and lets programmers choose their own modeling and meta-heuristic, thus supporting a wide variety of local search algorithms.

It is also worth emphasizing that the architecture builds on fundamental research in programming languages. It integrates one-way constraints pioneered in the Sketchpad and ThingLab object-oriented systems [24, 4] and generalizes them to accommodate finite differencing techniques on algebraic and set expressions [19, 29]. It also encapsulates efficient incremental graph algorithms [21, 1], and uses polymorphism heavily to obtain its compositionality. Observe also that the resulting architecture has some flavor of aspect-oriented programming [13], since constraints represent and maintain properties across a wide range of objects. Finally, note also that many of the concepts introduced in COMET are much more widely applicable and would benefit many other applications, where incremental algorithms and data structures are heavily used. This is typical in many greedy algorithms, as well as in many heuristic and approximation algorithms.

This paper illustrates the architecture using COMET, a Java-like programming language under development at Brown University. COMET supports the local search architecture with a number of novel concepts, abstractions, and control structures. However, it is important to point out that the architecture could be implemented equally well as a library or on top of an existing language. COMET simply enables us to experiment easily with a variety of designs and implementation techniques, and to choose a syntax reflecting the semantics of the architecture.

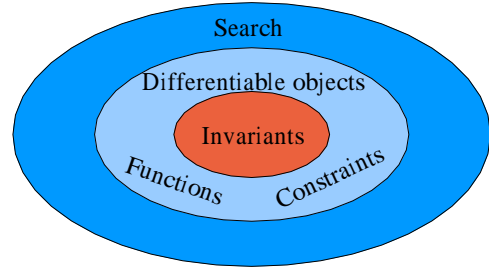


Figure 1: The Constraint-Based Architecture.

To demonstrate the feasibility and benefits of the architecture, we present several local search algorithms in COMET. Each of these applications can be expressed very concisely and the paper contains almost the entire code for each of them. Moreover, some of these applications are rather sophisticated and, in one case, COMET enabled us to close an open problem. We also report that the efficiency of a JIT implementation of COMET is comparable to special-purpose algorithms. It is also useful to mention that COMET helped us to design the fastest algorithm for warehouse location, which will be reported in another paper. As a consequence, we believe that the architecture brings significant benefits for the implementation of local search algorithms, which are so fundamental in numerous application areas.

The rest of the paper is organized as follows. Section 2 reviews the architecture in more detail. Section 3, 4, and 5 present three applications to give some of the flavor of the architecture. Section 6 describes the implementation, which generalizes finite-differencing techniques, and Section 7 concludes the paper.

2. THE ARCHITECTURE

Our architecture for local search, depicted in Figure 1, consists of a declarative and a search component organized in three layers. The kernel of the architecture is the concept of *invariants* (or one-way constraints) over algebraic and set expressions [15]. Invariants are expressed in terms of incremental variables and specify a relation which must be maintained under assignments of new values to its variables. For instance, the code fragment

```
inc{int} s(m) <- sum(i in 1..10) a[i];
```

declares an incremental variable s of type `int` (in a model m) and an invariant specifying that s is always the summation of $a[1], \dots, a[10]$. Each time, a new value is assigned to an element $a[i]$, the value of s is updated accordingly (in constant time). Note that the invariant specifies the relation to be maintained incrementally, not how to update it. Incremental variables are always associated with a model (m in this case), which makes it possible to use a very efficient implementation which dynamically determines a topological order in which to update the invariants. As we will see later in the paper, COMET supports a wide variety of algebraic and set invariants. It also contains a number of graph invariants (to incrementally maintain shortest and longest paths) but these are not discussed in this paper.

```

abstract class Constraint {
    inc{int} true();
    inc{int} violationDegree();
    inc{int} violations(inc{int} var);
    int getAssignDelta(inc{int} var,int val);
    int getSwapDelta(inc{int} v,inc{int} w);
    ...
}

```

Figure 2: The Abstract Class Constraint

Once invariants are available, it becomes natural to support the concept of *differentiable objects*, a fundamental abstraction for local search programming. *Differentiable objects maintain a number of properties (using invariants) and can be queried to evaluate the effect of local moves on these properties.* They are fundamental because many local search algorithms evaluate the effect of various moves before selecting the neighbor to visit. Two important classes of differentiable objects are constraints and functions. A differentiable constraint maintains properties such as its satisfiability, its violation degree, and how much each of its underlying variables contribute to the violations. It can be queried to evaluate the effect of local moves (e.g., assignments and swaps) on these properties. A differentiable function maintains the value of a (possibly) complex function and can also be queried to evaluate the variation in the function value under local changes.

Differentiable objects capture combinatorial substructures arising in many applications and are appealing for two main reasons. On the one hand, they are high-level modeling tools which can be composed naturally to build complex local search algorithms. As such, they bring into local search some of the nice properties of modern constraint satisfaction systems. On the other hand, they are amenable to efficient incremental algorithms that exploit their combinatorial properties. The use of combinatorial constraints is implicitly emerging from a number of research projects: It was mentioned in [28] as future research and used, for instance, in [5, 10, 18] as building blocks for satisfiability problems. Combinatorial functions play a similar role for optimization problems, where it is important to evaluate the variation of complex objective functions efficiently.

The `AllDifferent` constraint is an example of differential object. In its simplest form, `AllDifferent(a)` receives an array `a` of incremental variables and holds if all its variables are given different values. The `AllDifferent` constraint maintains a variety of properties incrementally. They include its violation degree, i.e., how many constraints `a[i] ≠ a[j]` are violated, as well as the set of variables which occur in such violations. Observe that the `AllDifferent` constraint captures a combinatorial substructure, i.e., a bipartite matching, which arises in many applications.

In COMET, differentiable constraints are objects implementing the abstract class `Constraint`, an excerpt of which is shown in Figure 2. The first three methods give access to incremental variables that maintain important properties of the constraint: its satisfiability, its violation degree, and the violations of its variables. The other two methods shown in Figure 2 are particularly interesting as they evaluate the

effect of assignments and swaps on the violation degree of the constraint. Method `getAssignDelta(var, val)` returns the variation of the violation degree when variable `var` is assigned value `val`. Similarly, method `getSwapDelta(v,w)` returns the variation of the violation degree when the values of variables `v` and `w` are swapped. Although this paper presents several differentiable constraints available in COMET, observe that COMET is an open language where users can add their own constraints by subclassing `Constraint`. (Similar principles and considerations apply to differentiable functions.)

Constraints can be composed naturally using *Constraints systems*, another fundamental abstraction of the architecture. A constraint system, which is itself a constraint, groups a collection of constraints together and maintains the satisfiability and violation degree of the set incrementally. The violation degree of a set of constraints is the summation of the violation degrees of its constraints. Being constraints themselves, constraint systems are differentiable and can also be queried to evaluate the effect of local moves. Constraint systems make local search algorithms more *compositional* and easier to extend. In particular, they allow new constraints to be added in the declarative component of an algorithm without changing the search component.

These first two layers, invariants and differentiable objects, constitute the declarative component of the architecture. The third layer of the architecture is the search component which aims at simplifying the implementation of heuristics and meta-heuristics, another critical aspect of local search algorithms. It does not prescribe any specific heuristic or meta-heuristic. Rather, it features high-level constructs and abstractions to simplify the neighborhood exploration and the implementation of meta-heuristics. The search layer includes several interesting control structures, such as selectors, abstractions to manipulate solutions, and advanced simulation techniques. This paper mostly uses selectors of the form

```
selectMin(j in S)(E)
```

which select the value `j` in `S` minimizing `E`. Selectors in COMET break ties randomly, since this is often desirable in local search applications. More advanced features not described here includes the ability to look ahead in the future, a feature pioneered by Kernighan and Lin [12].

3. THE QUEENS PROBLEM

This section describes a local search algorithm for the n -queens problem, i.e., the problem of placing n queens on a $n \times n$ chessboard so that no two queens attack each other. The simplicity of n -queens problem makes it an excellent vehicle to introduce many aspects of COMET.

The local search algorithm is based on the min-conflict heuristic [17]. 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. 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`.

```

int n = 512;
range Size = 1..n;
Model m();
UniformDistribution distr(Size);

inc{int} queen[i in Size](m,Size) := distr.get();
int neg[i in Size] = -i;
int pos[i in Size] = i;

ConstraintSystem S(m);
S.post(new AllDifferent(queen));
S.post(new AllDifferent(queen,neg));
S.post(new AllDifferent(queen,pos));
inc{set{int}} conflictSet(m);
conflictSet <- argMax(q in Size) S.violations(queen[q]);
m.close();

while (S.violationDegree())
  select(q in conflictSet)
    selectMin(v in Size)(S.getAssignDelta(queen[q],v))
      queen[q] := v;

```

Figure 3: The Queens Problem in Comet.

Figure 3 depicts the COMET program for the queens problem. As can easily be seen, the COMET program is extremely compact and high-level. Since this is our first application, we describe the program in detail. The first instructions declare the size of the board and a range containing all the board positions. The instruction

```
Model m();
```

declares a model `m`. As mentioned earlier, models are container objects which store incremental variables, invariants, and constraints. They make it possible to have an efficient planning/execution implementation based on dynamic topological orderings [16]. The above instruction is, in fact, syntactic sugar for the more traditional (Java-like) code

```
Model m = new Model();
```

The instruction

```
UniformDistribution distr(Size);
```

declares a uniform distribution which can be used to generate pseudo-random numbers in range `Size`. This random distribution will be used to generate the initial positions of the queens. The next instruction

```
inc{int} queen[i in Size](m,Size) := distr.get();
```

is particularly interesting. It declares an array of incremental variables `queen`, each variable `queen[c]` representing the row where the queen in column `c` is located. These incremental variables take their values in range `Size` and are initialized with random positions. Incremental variables are central in the architecture. They are used in invariants and constraints, and changes to their values induce a propagation step that updates all invariants and constraints directly or indirectly affected by the changes. Observe that the above instruction is rather compact. In fact, it is just syntactic sugar for the code

```

inc{int} queen[] = new inc{int}[Size];
forall(i in Size) {
  queen[i] = new inc{int}(m,Size);
  queen[i] := distr.get();
}

```

which is closer to traditional object-oriented code. Observe the use of the assignment operator `:=`, which assigns a value of type `T` to an incremental variable of type `inc{T}`. By contrast, the operator `=` assigns references.

3.1 The Declarative Component

The next couple of instructions are fundamental and specify the declarative component of the program. They describe the problem constraints in a high-level declarative way. The instruction

```
ConstraintSystem S(m);
```

declares a constraint system `S`, while the instructions

```

S.post(new AllDifferent(queen));
S.post(new AllDifferent(queen,neg));
S.post(new AllDifferent(queen,pos));

```

add the problem constraints to `S`. The first constraint specifies that all queens must be placed on different rows and is simply the `AllDifferent` differentiable object presented earlier. The next two constraints specify that the queens must be placed on different diagonals. They use a more general form of the constraint. More precisely, a constraint `AllDifferent(v,o)` holds if `v` and `o` are arrays whose range is `1..n` and the constraints

$$v[i] + o[i] \neq v[j] + o[j]$$

hold for all $1 \leq i < j \leq n$. Observe that these constraints specify the problem declaratively in a very natural way and capture a combinatorial substructure often used in combinatorial optimization problems.

The final instructions of the declarative component

```

inc{set{int}} conflictSet(m);
conflictSet <- argMax(q in Size) S.violations(queen[q]);

```

are particularly interesting as well. The first instruction declares an incremental variable `conflictSet` whose values are of type “set of integers”. The second instruction imposes an invariant ensuring that the values of `conflictSet` always be the set of queens that violate the most constraints. Observe that `S.violations(queen[q])` returns an incremental variable representing the number of violations in `S`. Operator `argMax(i in S) E` simply returns the set of values `v` in `S` which maximizes `E`. The last instruction

```
m.close()
```

closes the model, which enables COMET to construct a dependency graph to update the constraints and invariants under changes to the incremental variables.

Observe that the declarative component only specifies the properties of the solutions, as well as the data structures to maintain. It does not specify how to update the constraints, e.g., the violations of the variables, or how to update the conflict set. These are performed by the COMET runtime system, which uses optimal incremental algorithms in this case.

Size	State(s)	Solve(s)	Total(s)	Iter
8	0.01	0.02	0.03	249
16	0.01	0.00	0.01	21
32	0.02	0.01	0.03	56
64	0.04	0.02	0.06	80
128	0.10	0.08	0.18	238
256	0.17	0.12	0.29	206
512	0.35	0.37	0.72	304
1024	0.71	1.64	2.35	628
2048	1.43	6.07	7.50	1100
4096	2.95	22.56	25.51	2092
8192	6.16	85.91	92.07	4040
16384	14.69	333.89	348.58	7968
32768	42.10	1320.43	1362.53	15899

Table 1: Experimental Results for N-Queens.

3.2 The Search Component

The final part of the program

```
while (S.violationDegree())
  select(q in conflictSet)
    selectMin(v in Size)(S.getAssignDelta(queen[q],v))
      queen[q] := v;
```

specifies the search strategy. It iterates until the violation degree of the system is zero, meaning that all constraints are satisfied. Each iteration randomly selects a queen q in the conflict set, select a value v for queen q that minimizes the number of violations, and assigns v to `queen[q]`. Particularly interesting is the instruction

```
selectMin(v in Size)(S.getAssignDelta(queen[q],v))
```

which selects the value v minimizing the number of conflicts of queen q . It uses the ability to query the constraint system to find out the impact of assigning v to queen q . This query can be performed in constant time, thanks to the invariants maintained in each of the constraints.

There are a couple of observations to make at this point. First, observe that the search and declarative components are clearly separated in the program. It is thus easy to modify one of them without affecting the other. For instance, it is possible to add new constraints to the constraint system without any need to change the search component. Similarly, it is easy to replace the search component without affecting the declarative component. Although the two components are physically separated in the program code, they closely collaborate during execution. The search component uses the declarative component to guide the search, while the assignment `queen[q] := v` starts a propagation phase which updates all invariants and constraints. This compositionality and clear separation of concerns are some of the appealing features of the architecture. Second, observe the high-level nature of the resulting program. The declarative component is close to a formal specification of the problem, while the search component is a very abstract description of the search strategy. In fact, it can be observed that this search component is not at all problem-specific and could almost be reused as such in other contexts.

3.3 Experimental Results

Table 1 reports some experimental results for the program. The program runs in $O(n)$ space and each iteration takes time $O(n)$, since it is necessary to find the value minimizing the number of conflicts. In general, the propagation step takes constant time because the conflict set is small and does not change very much. The table reports the total CPU time in seconds on a 1.1Ghz PC running Linux, as well as the time for stating the constraints and for the search. It also reports the number of iterations. The results clearly indicate that the implementation is extremely competitive with low-level encodings of the algorithm [5] and is quadratic in time experimentally.

4. CAR SEQUENCING

Our second application is the car-sequencing problem, a challenging combinatorial optimization application [20, 7]. The main goal of this example is to produce evidence that hard combinatorial optimization are amenable to effective solutions within the architecture. Once again, the problem statement is short and concise, and it contains a differentiable object for a combinatorial substructure that arises in many applications. The resulting COMET program closed an open problem and is extremely efficient compared to other approaches.

4.1 The Problem

The car-sequencing problem consists of sequencing n cars on an assembly line so that the capacity constraints of all options are satisfied. The capacity constraints are of the form “ l out of u ” and specify that, out of u successive cars in the assembly line, no more than l may require the option. The input of the problem specifies the type of cars to be produced (called configurations), how many needs to be produced, and the capacity on the options. Each type of car is specified by a set of options (e.g., sunroof, CD player) it requires.

4.2 The Comet Program

Figure 4 depicts the core of the Comet program. It assumes that the data has been read and preprocessed, which takes about 20 lines of code and is not shown here. The fragment simply assumes that ranges `Configs`, `Cars`, and `Options` specify the types of cars to sequence, the set of cars, and their possible options. For every car c , the array element `car[c]` specifies the configuration of car c . For every option o , array element `option[o]` denotes the set of cars requiring o , while `lb[o]` and `ub[o]` specify its capacity constraint, i.e., they specify that no more than `lb[o]` cars in each sequence of size `ub[o]` may take the option.

The local search algorithm is a tabu search procedure that minimizes the number of violations of the capacity constraints until a valid sequencing is found. Its basic idea is to select the most violated car and to swap it with the car minimizing the number of conflicts.

4.3 The Declarative Component

The declarative component of the algorithm states the capacity constraints. More specifically, the declarative component of the algorithm declares an array `line` of incremental variables: `line[i]` denotes the car assigned to slot i in the

```

...
int tbl = 10;
int stableLimit = 300;
int nbDiversifications = 3;
range Diversifications = 1..nbDiversifications;

Model m();
RandomPermutation perm(Cars);
inc{int} line[c in Cars](m,Configs) := cars[perm.get()];
ConstraintSystem S(m);
forall(o in Options)
  S.post(new SequenceAtmost(line,options[o],lb[o],ub[o]));
inc{int} violations = S.violationDegree();
m.close();

int it = 0; int stable = 0;
int best = violations;
int tabu[Cars,Cars] = -1; int tabuLength = 2;
while (violations) {
  int old = violations;
  selectMax(c in Cars)(S.violations(line[c]))
  selectMin(v in Cars :
    line[c] != line[v] && tabu[c,v] <= it)
    (S.getSwapDelta(line[c],line[v])) {
    line[c] := line[v];
    if (violations >= old) {
      tabu[c,v] = it + tabuLength;
      tabu[v,c] = it + tabuLength;
    }
  }
  if (violations < old && tabuLength > 2)
    tabuLength = tabuLength - 1;
  if (violations >= old && tabuLength < tbl)
    tabuLength = tabuLength + 1;
  if (violations < best) {
    stable = 0;
    best = violations;
  } else {
    if (stable == stableLimit) {
      stable = 0;
      forall(i in Diversifications)
        select(c in Cars, v in Cars: c != v)
          line[c] := line[v];
      best = violations;
    } else
      stable = stable + 1;
  }
  it = it + 1;
}

```

Figure 4: Car Sequencing in Comet

assembly line. It then declares the constraint system S and posts all capacity constraints in S . The capacity constraints are of the form `SequenceAtmost(car,0,n,m)` and they hold if every sequence `car[i], ..., car[i+m]` has atmost n cars in 0 . These constraints are differentiable and they capture a combinatorial subproblem that arise in many sequencing and scheduling applications, including sport-scheduling problems. The violation degree of such a constraint is the number of sequences `car[i], ..., car[i+m]` that have more than n cars in 0 . It is implemented using a variety of non-trivial invariants.

The declarative part also declares an incremental variable `violations` which maintains the violation degree of S . Observe again the simplicity and high-level nature of the declarative component.

Problem	4	8	26	41	21a
Time(Min)	0.41	2.3	1.02	1.31	0.91
Time(Max)	0.64	56.05	21.21	20.85	3.27
Time(Av.)	0.497	25.765	5.057	7.126	1.935

Table 2: Experimental Results for Car-Sequencing.

4.4 The Search Component

The search component is a variable-length tabu search with a diversification element. Its key idea is to select a car appearing in the most conflicts and to swap it with the car which leads to the least violations. This basic idea is implemented by the instructions

```

selectMax(c in Cars)(S.violations(line[c]))
  selectMin(c in Cars : line[c] != line[v])
    (S.getSwapDelta(line[c],line[v]))
  line[c] := line[v];

```

which are the core of the search component. (The actual instructions in fact also contain the tabu search test.) Observe how the `selectMin` instruction queries the constraint system with method `S.getSwapDelta(car[c],car[v])` to evaluate the effect of swapping cars `car[c]` and `car[v]`. Once the two cars are selected, instruction `car[c] := car[v]` performs the swap and updates the invariants and differentiable objects. The rest of the search component is relatively simple. It uses a tabu list whose length may be updated at each iteration and it diversifies the search once the number of violations has not been improved for a number of iterations. The diversification simply consists of swapping a number of cars randomly.

4.5 Experimental Results

Table 2 gives the experimental results. We took the satisfiable instances from the CSP lib library [11]. These instances were generated by B. Smith and are very difficult in general (Problem 26 was open). We also took a satisfiable version of Problem 21 which is also open. The algorithm solved all satisfiable instances and closes Problem 26. We report average, minimum, and maximum CPU Time in seconds over 10 runs on a 800mhz PC running Linux. The COMET program is extremely effective on this problem and is probably one of the most efficient algorithms available for this problem at this point. It is important to stress that this performance was obtained by a concise and high-level program.

5. THE PROGRESSIVE PARTY PROBLEM

Our final example is a COMET program for the progressive party (PP) problem, which is interesting for several reasons. On the one hand, the PP problem is often used as a benchmark for comparing approaches and algorithms in combinatorial optimization, since it is non-trivial. On the other hand, it features a variety of heterogeneous constraints which makes it an interesting exercise in modeling. The COMET program illustrates the use of a constraint system and combinatorial constraints once again. It also features a tabu search with an intensification component.

5.1 The Problem

The progressive party problem can be described as follows (as is traditional, we assume that host boats have been selected). We are given a set of boats (the hosts), a set of

```

Model m();
UniformDistribution distr(Hosts);
inc{int} boat[Guests,Periods](m,Hosts) := distr.get();
ConstraintSystem S(m);
forall(g in Guests) {
  inc{int} r[p in Periods] = boat[g,p];
  S.post(new AllDifferent(r),2);
}
forall(p in Periods) {
  inc{int} c[g in Guests] = boat[g,p];
  S.post(new WeightedAtmost(c,crew,cap),2);
}
forall(i in Guests, j in Guests : j > i) {
  inc{int} ri[p in Periods] = boat[i,p];
  inc{int} rj[p in Periods] = boat[j,p];
  S.post(new MeetAtmost(ri,rj,1));
}
inc{int} vd = S.violationDegree();
m.close();

int tabuLength = 2;
int tabu[Guests,Periods,Hosts] = -1;
Solution solution(m); int best = vd;
int it = 0; int stable = 0;
int stableLimit = 2000;
while (vd) {
  int old = vd;
  selectMax(g in Guests,p in Periods)
    (S.violations(boat[g,p]))
  selectMin(h in Hosts: tabu[g,p,h] <= it ||
    S.getAssignDelta(boat[g,p],h) + vd < best)
    (S.getAssignDelta(boat[g,p],h)) {
    tabu[g,p,boat[g,p]] = it + tabuLength;
    boat[g,p] := h;
    if (vd < old && tabuLength > 2)
      tabuLength = tabuLength - 1;
    if (vd >= old && tabuLength < tbl)
      tabuLength = tabuLength + 1;
  }
  if (vd < best) {
    best = vd; stable = 0;
    solution = new Solution(m);
  } else {
    if (stable == stableLimit) {
      solution.restore();
      stable = 0; it = it + tabuLength;
    }
    it = it + 1; stable = stable + 1;
  }
}
}

```

Figure 5: The Progressive Party Problem in Comet

crews (the guests), and a number of time periods. Each host has a limited capacity and each crew has a given size. The problem consists of assigning a host to every guest for each time period in order to satisfy three sets of constraints. First, the capacity of the hosts cannot be exceeded by the guest crews. Second, every guest must visit a different host at each time period. Finally, no two guests can meet more than once over the course of the party.

Figure 5 depicts the core of the COMET program. It receives as inputs two ranges `Hosts` and `Guests` that denote the set of hosts and the set of guests respectively. In addition, it receives two arrays of integers `cap` and `crew` where `cap[h]` is the capacity of host `h` and `crew[g]` is the size of guest `g`. Once again, the local search algorithm for the PP problem features a declarative and a search component.

5.2 The Declarative Component

The declarative component primarily declares the incremental variables and the problem constraints. It features a variety of combinatorial constraints and illustrates how to associate weights with constraints, a common technique in local search (e.g., [23]). The instructions

```

inc{int} r[p in Periods] = boat[g,p];
S.post(new AllDifferent(r),2);

```

specify that a guest `g` never visits a host more than once over the course of the party. Observe that the `post` method which specifies a weight of 2 for the constraint. The instructions

```

inc{int} c[g in Guests] = boat[g,p];
S.post(new WeightedAtmost(c,crew,cap),2);

```

specify the capacity constraints for period `p`. Constraint

```
WeightedAtmost(c,crew,cap)
```

is a generalized cardinality constraint [27, 3] which holds if

$$\sum_{i \in \text{Guests}} \text{crew}[i] * (c[i] = j) \leq \text{cap}[j]$$

for all $j \in \text{Hosts}$. Once again, this constraint is differentiable and captures a subproblem arising in many applications. Finally, the instructions

```

inc{int} ri[p in Periods] = boat[i,p];
inc{int} rj[p in Periods] = boat[j,p];
S.post(new MeetAtmost(ri,rj,1));

```

specify that no two guest crews meet more than once. The constraint `meetAtmost(a,b,1)` is also a cardinality constraint which holds if

$$\#\{i \in R \mid a[i] = b[i]\} \leq l$$

where R is the range of arrays `a` and `b`.

5.3 The Search Component

The search component is a tabu-search procedure minimizing the violation degree of the constraint system. It first selects a guest crew `g` and a time period `t` such that variable `boat[g,t]` appears in the most conflicts. It then selects a non-tabu host for the pair `(g,t)` that minimizes the violation degree of the system. The host selection features an aspiration criteria

```
S.getAssignDelta(boat[g,p],h) + vd < best
```

which overrides the tabu status when the assignment improves upon the best “solution” found so far.

The search also includes an *intensification* process. The key idea behind the intensification is to restart from the best found solution when the number of violations has not decreased for a long time. The search uses the solution concept of COMET for implementing the intensification in a simple manner. The instruction

```
Solution solution(m);
```

declares a solution which stores the current value of all decision variables, i.e., incremental variables whose values are not specified by an invariant. Each time a better solution is found, the instruction

Hosts/Periods	6	7	8	9	10
1-12,16	0.98	1.64	5.13	90.19	
1-13	0.61	0.90	1.17	4.41	21.00
1,3-13,19	0.90	1.53	5.28	253.92	
3-13,25,26	1.21	1.81	7.02	82.66	
1-11,19,21	4.50	24.35			
1-9,16-19	6.20	161.16			

Table 3: Experimental Results for the PP Problem.

```
solution = new Solution(m);
```

makes sure that `solution` now maintains the best solution found so far. After a number of iterations without improvements, the instruction `solution.restore()` restores the best available solution and resumes the search from this state. Once again, observe the simplicity of the search component which can be described very concisely.

5.4 Experimental Results

Before presenting the results, it is useful to emphasize that the program provides a very high-level and natural modeling of the application. Yet, as the results show, the algorithm compares extremely well with low-level codes developed for this application. The experiments use the same host configurations as in [28] to evaluate the algorithm. For each of these configurations, we consider problems with 6 periods (as in [28]) but also with 7, 8, 9, and 10 time periods. Table 3 reports median CPU Time in seconds for 10 runs of the algorithm. As can be seen, the algorithms easily outperforms the approach in [28].

6. IMPLEMENTATION

This section gives an overview of the implementation. The COMET system includes both an interpreter and a just-in-time compiler, both of which are based on the same abstract machine and runtime system. Most of this technology is standard. As a consequence, we focus exclusively on the novel aspects of runtime systems: invariants and differentiable objects.

6.1 Invariants

COMET supports a wide variety of invariants over algebraic, set, and graph structures. These invariants are often constructed with aggregate operators such as summations, products, unions, intersections, and projections. When a model is closed, its invariants are normalized into primitive invariants.

The key implementation issue is how to propagate changes to incremental variables efficiently. COMET uses an incremental planning/execution algorithm which generalizes finite-differencing techniques. The planning phase determines a suitable order to propagate the changes, while the execution phase carries out the propagation. The implementation guarantees that every pair $\langle \text{variable}, \text{invariant} \rangle$ is propagated at most once during a propagation cycle.

Invariants are classified into two main categories: static and dynamic invariants. Static invariants such as

```
x <- y * z
```

are well-studied in the programming language community (e.g., [19, 29]). These invariants are appealing because they make the dependencies between their variables explicit. As a consequence, it is simple in general to find a partial ordering for the propagation steps that guarantees that each pair $\langle \text{variable}, \text{invariant} \rangle$ is considered at most once. This assumes, of course, that there are no cycle in the dependency graph, which is the case in combinatorial optimization applications.

Dynamic invariants are one of the novelties of the architecture and they generalize finite differencing techniques to more dynamic contexts. Indeed, it often happens in combinatorial optimization that the dependency graph seems to have cycles because the dependencies are not known precisely when a model is closed. Consider a collection of invariants of the form

```
start[i] <- start[prec[i]] + duration[prec[i]]
```

which arise in job-shop scheduling applications. Here `prec[i]` is an incremental variable that denotes the predecessor of job `i`, while `start[i]` denotes the starting time of `i`. Since the value of `prec[i]` evolves during the computation, it is necessary to include dependencies between `start[i]` and all its possible predecessors

```
start[1], ..., start[n].
```

This creates cycles in the dependency graph, although, at execution time, none of these cycles actually materializes. (Cycles could actually appear if programmers make mistakes and COMET would raise an error in those cases.) We call these invariants *dynamic*, because determining the actual dependencies requires knowledge of the updated values of some variables, in addition to the structure of the expressions. More precisely, consider the typical normalized dynamic invariant

```
x <- element(y, x1, ..., xn)
```

which specifies that $x = x_y$, where x, y, x_1, \dots, x_n are incremental variables. Clearly, at any given computation time, the value of x depends on the values of y and exactly one of the x_i . However, the actual x_i which variable x depends on is only known when y is updated.

As a consequence, when dynamic invariants are present, it is not possible to deduce a priority ordering. However, not allowing such invariants would dramatically reduce the expressiveness of the architecture and would make it impossible to state many interesting applications. The key observation is to recognize that, once some of the variables are updated, it becomes easy to find a priority ordering, or to detect that none exists.

In order to accommodate dynamic invariants, the implementation decomposes the dependency graph in strongly connected components (SCCs). A priority ordering for these SSCs is computed once, when the model is closed. The execution phase now follows the priority ordering to propagate the invariants. When a SCC is considered, the values of all incremental variables in previous SCCs have been updated and hence the invariants in the SCC become static. As a consequence, it is now possible to find a priority ordering

```

procedure execute( $\mathcal{I}$ )
begin
   $\langle \mathcal{I}_0, \dots, \mathcal{I}_p \rangle := \text{serialize}(\mathcal{I});$ 
  for(  $i := 0; i \leq p; i++$ ) do
     $t := \text{plan}(\mathcal{I}_i);$ 
    execute( $\mathcal{I}_i, t$ );
  endfor
end

```

Figure 6: The Propagation Algorithm

for the SSC and the execution phase guarantees that each pair $\langle \text{variable}, \text{invariant} \rangle$ is considered at most once.

The overall computational model is depicted in Figure 6. Given a set of invariants \mathcal{I} , the algorithm serializes them, by identifying the SSCs and finding a priority ordering. Note that there may be several “equivalent” priority orderings but it does not actually matter which one is selected. The algorithm then considers each SCC individually, finds a priority ordering using the values of the variables in the predecessor SCCs, and updates the invariants of the SCC. The actual update is rather standard in general: It simply uses finite differencing techniques.

It is important to stress that COMET uses an incremental version of this algorithm. The serialization takes place once when the model is closed. Moreover, the priority ordering for each SCC is not recomputed from scratch at each change. Instead, COMET uses the incremental algorithm of Alpern et al. [1]

6.2 Differentiable Objects

Differentiable objects are generally implemented using a collection of invariants, as well as specific algorithms which exploit the combinatorial structure and the invariants to evaluate the effect of changes. We illustrate such an implementation using the simplest constraint mentioned in the paper: `AllDifferent(a[1], ..., a[n])`.

The basic idea behind the implementation is to maintain invariants of the form

```
o[v] <- #{ i in 1..n | a[i] = v };
```

for all values v in D , the domains of values that the $a[i]$ can take. Informally speaking, variable $o[v]$ denotes the number of times value v occurs in the array. These invariants can be maintained globally very efficiently, i.e., in time $O(\Delta(a))$, where $\Delta(a)$ denotes the number of variables in $a[1], \dots, a[n]$ whose values have changed. Once these invariants are stated, it is easy to define the satisfiability of the constraint as

```
satisfiable <- and(v in D) (o[v] <= 1);
```

Once again, this invariant can be maintained efficiently. Note also that the number of violations of a variable v is given by

```
vi[v] <- max(0, o[v]-1);
```

The effect of assigning value val to variable var is given by the expression

```

if (var == val) then
  0
else
  o[val] + max(0, o[var]-2) - (vi[val] + vi[var])

```

The overall implementation is rather simple in this case. However, many other constraints are much more challenging and rely on fundamental properties of the combinatorial structure that the constraint captures.

7. CONCLUSION

The research described in this paper was motivated by our desire to reduce the development time of local search algorithms. Local search is one of the fundamental approaches to combinatorial optimization, but efficient algorithms are, in general, difficult to implement. Indeed, local search programming often involves complex data structures and algorithms and requires considerable experimentation.

This paper has proposed a constraint-based, object-oriented, architecture for implementing various classes of local search algorithms. The architecture consists of declarative and search components. The declarative component includes *invariants*, which maintain complex expressions incrementally, and *differentiable objects*, which maintain properties that can be queried to evaluate the effect of local moves. Differentiable objects are high-level modeling concepts, such as constraints and functions, that capture combinatorial substructures arising in many applications. The search component supports various abstractions to specify heuristics and meta-heuristics. The architecture is implemented by a planning/execution scheme which generalizes finite-differencing techniques to the dynamic environment necessary for local search applications.

The paper illustrated the architecture with the language COMET and a variety of applications. The applications demonstrate that the architecture may reduce development time significantly. They also indicate that COMET makes it possible to write local search algorithms in terms of high-level modeling concepts and abstractions. The resulting programs are easier to read, modify, and extend, which is important in an area where experimentation is critical to identify algorithms which behave well in practice.

The experimental results also indicate that the architecture can be implemented to be competitive with low-level encodings of the algorithms.

Acknowledgments

Pascal Van Hentenryck is partially supported by NSF ITR Award DMI-0121495.

8. REFERENCES

- [1] B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 32–42, San Francisco, California, 22–24 Jan. 1990.
- [2] A. Andreatta. A framework for the development of local search heuristics with an application to the phylogeny problem, 1997.
- [3] N. Beldiceanu and M. Carlsson. Revisiting the cardinality Operator. In *ICLP-01*, Cyprus, 2001.
- [4] A. Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation

- Laboratory. *ACM Transaction on Programming Languages and Systems*, 3(4):353–387, 1981.
- [5] C. Codognet and D. Diaz. Yet Another Local Search Method for Constraint Solving. In *AAAI Fall Symposium on Using Uncertainty within Computation*, Cape Cod, MA., 2001.
- [6] A. Colmerauer. An Introduction to Prolog III. *Commun. ACM*, 28(4):412–418, 1990.
- [7] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the Car Sequencing Problem in Constraint Logic Programming. In *ECAI-88*, August 1988.
- [8] A. Fink, S. Voss, and D. Woodruff. Building reusable software components for heuristic search, 1998.
- [9] R. Fourer, D. Gay, and B. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press, San Francisco, CA, 1993.
- [10] P. Galinier and J.-K. Hao. A General Approach for Constraint Solving by Local Search. In *CP-AI-OR'00*, Paderborn, Germany, March 2000.
- [11] I. Gent and T. Walsh. CSPLib: a benchmark library for constraints. In *CP'99 (Short Paper)*, Alexandria, VA, October 1999.
- [12] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49:291–307, 1970.
- [13] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP '97*, pages 220–242. June 1997.
- [14] K. McAloon and C. Tretkoff. 2LP: Linear Programming and Logic Programming. In V. Saraswat and P. V. Hentenryck, editors, *Principles and Practice of Constraint Programming*. The MIT Press, Cambridge, Ma, 1995.
- [15] L. Michel and P. Van Hentenryck. Localizer: A Modeling Language for Local Search. *Informs Journal on Computing*, 11(1):1–14, 1999.
- [16] L. Michel and P. Van Hentenryck. Localizer. *Constraints*, 5:41–82, 2000.
- [17] S. Minton, M. Johnston, and A. Philips. Solving Large-Scale Constraint Satisfaction and Scheduling Problems using a Heuristic Repair Method. In *AAAI-90*, August 1990.
- [18] A. Nareyek. *Constraint-Based Agents*. Springer Verlag, 1998.
- [19] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.
- [20] B. Parrello, W. Kabat, and L. Vos. Job-Shop Scheduling Using Automated Reasoning: A Case Study of the Car-Sequencing Problem. *Journal of Automated Reasoning*, 2(1):1–42, 1986.
- [21] G. Ramalingam. *Bounded Incremental Computation*. PhD thesis, University of Wisconsin-Madison, 1993.
- [22] 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.
- [23] B. Selman, H. Kautz, and B. Cohen. Noise Strategies for Improving Local Search. In *AAAI-94*, pages 337–343, 1994.
- [24] I. Sutherland. SKETCHPAD: a Man-Machine Graphical Communication System. Cambridge, MA, MIT Lincoln Labs, 1963.
- [25] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series, The MIT Press, Cambridge, MA, 1989.
- [26] P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, Cambridge, Mass., 1999.
- [27] P. Van Hentenryck and Y. Deville. The Cardinality Operator: A New Logical Connective and its Application to Constraint Logic Programming. In *ICLP-91*, June 1991.
- [28] J. Walser. *Integer Optimization by Local Search*. Springer Verlag, 1998.
- [29] D. Yellin and S. R.E. INC: A Language for Incremental Computations. In *PLDI'88*, pages 115–124, Atlanta, Georgia, 22–24 June 1988. *SIGPLAN Notices* 23(7), July 1988.